Automated Search for Good Coverage Criteria: Moving from Code Coverage to Fault Coverage Through Search-Based Software Engineering

Phil McMinn[1] Mark Harman[2] Gordon Fraser[1] Gregory M. Kapfhammer[3]
[1]University of Sheffield, UK [2]University College London, UK [3]Allegheny College, USA

ABSTRACT

We propose to use Search-Based Software Engineering to automatically evolve coverage criteria that are well correlated with fault revelation, through the use of existing fault databases. We explain how problems of bloat and overfitting can be ameliorated in our approach, and show how this new method will yield insight into faults — as well as better guidance for Search-Based Software Testing.

1. INTRODUCTION

Since exhaustive testing is generally impossible, a variety of *coverage criteria* have been defined in the software testing literature to provide a systematic basis with which to select test cases. Typically, coverage criteria are defined in terms of a set of test requirements that mandate specific software elements are somehow "satisfied" or "covered" (i.e., executed) [1]. Among the most popular coverage criteria are code coverage criteria, which, for example, mandate that aspects of the software's code structure are executed as part of each test requirement — for instance, each program statement or branch. While these approaches might provide a method of dividing up a software system into components that need to be tested, evidence suggests that they are not as effective as they could be at revealing faults in real-world systems [6, 7, 10, 18].

This position paper contends that it is time to shift focus from searching for test cases that satisfy traditional coverage metrics to *searching for the coverage criteria themselves* — in particular, coverage criteria that are well-correlated with real fault revelation. Software testing research is maturing to the point at which real software fault repositories are becoming available (e.g., [3, 11]). Search-Based Software Testing (SBST), and Search-Based Software Engineering (SBSE) more generally, are therefore in an ideal position to leverage this information. Based on knowledge of real faults, existing SBSE approaches could be used to learn the essence of criteria that are good at revealing faults; SBSE for SBST.

While overfitting to a particular fault database could be mitigated by standard machine learning approaches, it may be an advantage in this circumstance: the learned criteria might yield insight into the types of faults prevalent in certain classes of software, or enlighten teams or individual programmers to the types of faults that they are prone to introducing when developing software.

SBST16, May 16-17, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2897010.2897013

2. TRADITIONAL COVERAGE CRITERIA AND FAULT COVERAGE

There are many ways in which traditional software coverage criteria fail to address the problem of real fault discovery. The types of faults that are *expected* to be found are generally unknown when applying a coverage criterion. Following their application, a tester does not know what types of fault may *remain* in the software. Finally, although there are some exceptions (e.g., [16]), they are typically limited to revealing faults of commission only, ignoring the whole class of defects resulting from faults of omission.

Conversely, electronic engineering has the explicit concept of a "fault model" and *fault coverage criteria*, which are directed at certain types of fault, such as "stuck-at" faults [14]. Stuck-at faults are manufacturing defects where individual signals and pins on a circuit board get "stuck" at a certain logical value that cannot be altered. While there has been work in the software testing literature on classifying faults and developing fault taxonomies (e.g., [4, 9, 12]), more generalized testing strategies are yet to emerge from them. The question, therefore, is how we might develop something like a "fault coverage" criterion for software testing?

3. EVOLVING COVERAGE CRITERIA TO TARGET REAL FAULTS

Our position is to use Search-Based Software Engineering to "learn" (i.e., search for) coverage criteria from databases of software faults, such that the learned coverage criteria are closely correlated with fault revelation. We now discuss the components of such an approach and the ways in which such a vision may be achieved.

Fault Database. The first issue is the need for a fault database that catalogs real faults. Potential candidates include Defects4J [11] and CoREBench [3]. Defects4J is a collection of over 350 faults for five open source Java programs, and archives not only defective program versions, but also their fixes and test suites; CoREBench logs 70 regression errors for four open source C++ programs.

Fitness Function. The search algorithm would learn "good" criteria where "goodness" (i.e., fitness) is defined as the correlation between greater achievement of coverage and greater fault revelation. Since we can measure statistical correlation in a number of ways, this immediately suggests candidate fitness functions for coverage criteria evolution. At the core of the fitness function, however, is a need to measure the fault-finding capability of a criterion from a collection of sample test suites. Since these test suites will not exist in advance, they must be automatically generated.

Generation of Sample Test Suites. To evaluate a candidate criterion the proposed process could generate test suites that fulfill the criteria to varying degrees, with which we can measure and correlate fault-finding capability. This is already achievable with existing SBST techniques, that seek to measure the "distance" to the coverage of code-related test requirements [15], and for measuring non-functional properties such as timing behavior.

This mechanism could prove to be an expensive process, however, requiring the generation of a set of test suites just to evaluate a single candidate criterion. Instead, the problem could be tackled from the opposite direction: the up-front generation of a suitable "universe" of test suites with varying levels of fault exposure. These test suites could then be used to evaluate *all* candidate criteria, but this time measuring and correlating with the coverage of each individual criterion. We would then instead require distance metrics for explicitly executing and propagating faults in the database, in the vein of those proposed by Shamshiri et al. [17].

Representation of Criteria. Resolving how to represent coverage criteria is key to the success of the entire approach presented in this paper: if the "language" of coverage criteria underpinning the representation is not expressive enough, the technique will fail in its goal to learn good fault-finding criteria. A flexible structure for the representation would be the tree structures used in Genetic Programming (GP), which requires the definition of the terminal nodes that feature in the tree. We propose a scheme of "directives" and "operators": directives require certain activities, for instance the use of certain inputs, the observation of certain outputs, forms of non-functional behavior to be observed, or program structures that are to be executed by the test requirements of the adequacy criterion. Operators define the extent or degree to which directives are to be achieved, for example a certain percentage or diversity of program paths executed, extremes of program path (i.e., the shortest or longest); and ways in which those directives will be combined, for instance through the AND and OR logical operations.

An initial set of directives and operators could be devised solely from existing coverage criteria. Initial experiments could then seek to discover what combinations and factors of existing coverage criteria are well suited to revealing faults in the database. We would then seek to diversify this set to include new types of properties such as those mentioned previously (i.e., outputs and non-functional behavior relating to, for instance, program performance).

Handling Bloat. The generated coverage criteria may not be succinct, due to GP-bloat [2]. This could be ameliorated by using the delta-debugging algorithm [19] to simplify the generated criteria. Also, given that we would have a language for test requirement specification, we could define meaning-preserving transformation rules on it. Such rules could then be used to simplify the final product of the GP process to remove any unwanted bloat. Indeed, search-based transformation could be used to remove redundancy, as with previous work for programming languages [5].

Overfitting. Evolving coverage criteria from a set of given faults naturally raises the issue of overfitting; perhaps the evolved coverage criteria would be well correlated with faults in the given database, but not more generalizable. To address this we could use standard machine learning techniques, such as sampling, that have been found to avoid overfitting in other SBSE paradigms such as genetic improvement [13]. This also tends to reduce the computation cost of fitness evaluation, which may otherwise prove prohibitive. However, even when overfitting does occur, the evolved coverage criteria may be valuable to the software tester, as they may reveal insights about the nature of faults in the class of software systems to which the criteria are fitted; SBSE is not merely about finding solutions, but also about discovering and using insight [8].

Clustering. Not all faults and software systems are the same. There is a complex interplay between the systems and their characteristics, the faults they may contain, the failures they cause, and the test cases that act as witnesses. Hitherto, this complexity has simply been seen as a barrier to be overcome in automating test data generation. However, we believe that our approach may be used to shed light on this interplay, as distance measurement can be used for both test cases and coverage criteria. Clustering systems and faults according to these two distance measures may reveal insights into the relationship between systems, faults, tests and the criteria that seek to capture their relationships. Furthermore, by identifying the building blocks (or genetic components) that are common to all such clusters, we may approach the fundamental question of identifying the building blocks of what constitutes good test adequacy (at least for certain well-defined classes of systems and faults).

Relation to Mutation Testing. While killing all of the mutants of a mutation testing strategy is also a form of fault coverage, mutation analysis generally results in a lot of mutants. Coverage criteria are generally cheaper to apply. Furthermore, our evolved coverage criteria may also take account of further aspects of software than are currently covered by mutants designed to mimic real developer faults, for example non-functional properties like memory usage.

4. CONCLUSION

We propose to search for coverage criteria for SBST, using GP to evolve test requirements that are well correlated with fault revelation. Our coverage criteria will not only take into account program code, and its functional behavior, but also non-functional properties such as execution time. A key enabler for this work is the presence of a fault database that the GP process can use to learn about faults and formulate generalized criteria. Since fault databases are now starting to appear for open source programs in different languages, the time is nigh for the ideas we propose here to be undertaken.

5. REFERENCES

[1] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[2] L. Attenberg. Emergent phenomena in genetic programming. In *Proc. of Evolutionary Programming*, 1994.

[3] M. Böhme and A. Roychoudhury. CoREBench: Studying complexity of regression errors. In *Proc. of ISSTA*, 2014.

[4] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M. Y. Wong. Orthogonal defect classification — a concept for in-process measurements. *IEEE TSE*, 18(11), 1992.

[5] D. Fatiregun, M. Harman, and R. Hierons. Search-based amorphous slicing. In *Proc. of WCRE*, 2005.

[6] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg. Does automated unit test generation really help software testers? In *Proc. of ISSTA*, 2013.

[7] G. Gay, M. Staats, M. Whalen, and M. Heimdahl. The risks of coverage-directed test case generation. *IEEE TSE*, 2015.

[8] M. Harman. The current state and future of search based software engineering. In *Proc. of FOSE*, 2007.

[9] J. H. Hayes. Building a requirement fault taxonomy: Experiences from a NASA verification and validation research project. In *Proc. of ISSRE*, 2003.

[10] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proc. of ICSE*, 2014.

[11] R. Just, D. Jalali, and M. D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proc. of ISSTA*, 2014.

[12] D. R. Kuhn. Fault classes and error detection capability of specification-based testing. *ACM TOSEM*, 8(4), 1999.

[13] W. B. Langdon and M. Harman. Optimising existing software with genetic programming. *IEEE TEC*, 19(1), 2015.

[14] L. Lavagno, G. Martin, and L. Scheffer. *Electronic Design Automation For Integrated Circuits Handbook*. CRC Press.

[15] P. McMinn. Search-based software test data generation: A survey. *STVR*, 14(2), 2004.

[16] P. McMinn, C. J. Wright, and G. M. Kapfhammer. The effectiveness of test coverage criteria for relational database schema integrity constraints. *ACM TOSEM*, 25(1), 2015.

[17] S. Shamshiri, G. Fraser, P. McMinn, and A. Orso. Search-based propagation of regression faults in automated regression testing. In *Proc. of Regression Testing Workshop*, 2013.

[18] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri. Do automatically generated unit tests find real faults? An empirical study of effectiveness and challenges. In *Proc. of ASE*, 2015.

[19] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE TSE*, 28(2), 2002.