Socio-technical Factors for Automated Test Generation

Iftekhar Ahmed
University of California, Irvine
iftekha@uci.edu

Alex Groce
Northern Arizona University
agroce@gmail.com

1 Abstract

This project proposes to investigate the applicability of *socio-technical* factors to automated test generation heuristics. The core insight is that *socio-technical* factors (e.g., statements involved in merge conflicts, statements exhibiting bad code smells, statements not following the review process etc.) have proven to be effective in predicting failures, so they should also be effective for guiding automated test case generation for both unbounded and limited time budgets. In the first phase of this project, we will investigate the applicability of various *socio-technical* factors that have not been used in automated test generation, although have significant impact on the bug-proneness of code. In the second phase, we will compare the effectiveness of different *socio-technical* factors (and traditionally-used purely technical source-code-related factors) in test case generation heuristics for the limited time budget test efforts essential for incremental testing.

2 Problem Statement

Testing is an essential technique for ensuring that software is robust and reliable. However, there has been an exponential growth in the complexity of software, and the cost of testing also has risen accordingly [9]. We cannot effectively test such complex systems using manually written test cases: as a result quality assurance must increasingly rely on improved automated test case generation [4, 7]. Various code properties have been used for generating test cases, including structural [11], functional [12], and non-functional properties [13]. The majority of these properties are technical in nature: rooted in the code itself or its specification; however, software development is not purely a technical activity, but a complex *socio-technical* activity with larger organizational goals and context, and structure related to process, not product. For example, statements involved in merge conflicts are not only related to the technical aspect of coding but also is related to task distribution process, hence a *socio-technical* factor. Development activity traces left behind in the code base, version control systems, issue trackers, and discussion forums allow us to understand the complex interactions happening during the development and researchers have recently started analyzing these interactions to predict which code is faulty. In our own work, we found that merge conflicts and design issues (a.k.a. "code smells") are better predictors of faulty code locations when used together rather than being used individually [1] . However, the applicability of these factors in automated test case generation is yet to be investigated. For example, while randomly generating test cases, instead of using equal selection probability, we can bias the selection towards methods that were involved in merge conflicts, methods with bad code smells and high code complexity and other combinations of traditional and *socio-technical* factors. Intuitively, these biased test generation techniques should be more effective, in that they include more context that influences code quality.

3 Research Plan

***Socio-technical* factors for test case generation:** Previous work on generating test cases investigated structural [11], functional [12], and non-functional [13] properties. None of these studies investigated the effectiveness of *socio-technical* factors in test case generation. We propose to produce the first such general, systematic, examination. We will begin by comparing state-of-the-art automated test generation systems, with and without use of socio-technical factors, on benchmarks such as Defects4J [8]. Because there are only 6 projects in the Defects4J benchmark, we will curate our own data-set combining information from sources such as bug tracking systems and patches submitted to the code base for fixing bugs using the techniques used in our prior work [2], applied to a large number of randomly selected, mature open source projects, and will release that as a benchmark. We will also use the existing manually written tests and augment them by generating additional tests using both state-of-the-art automated test generation systems and *socio-technical* factors (in conjunction, and separately) and compare test performance to identify the best way to generate an effective, efficient test suite.

***Socio-technical* factors for *limited-budget* test case generation:** In the second phase of the project, we will investigate the applicability of *socio-technical* factors for generating test cases for limited time budgets, where the test case generation and execution time are only proportional to the *size of a change* instead of the *whole code base*. Limited budget test generation is more demanding then traditional testing, in part because limited-budget automated test generation should only require minimal additional computational effort compared to pure random testing, if it is to obtain satisfactory results for very small budgets. Given these constraints, *socio-technical* factors are strong candidates for exploration as they do not require any complex infrastructure and add minimal overhead to test generation. As with traditional automated test generation, previous investigations of limited-budget test generation [6] have focused on using technical (code) factors only to guide testing. We propose to perform the first systematic investigation of *socio-technical* factors in this context as well, using similar methods as with our first study. We will investigate the effectiveness for different time budgets using average percentage faults detected (APFD) [5] curved for both known faults and seeded faults from mutation testing [3].

Concretely, the basic experimental effort in both thrusts would resemble the effort of Shamshiri et. al [10], scaled up to many more subject programs in more languages, and focused on how multiple tools are impacted by heuristics that change the way "select a random method call" (a foundational element in almost all test generation tools) is performed. For example, how does equal-probability random selection (the usual approach in random and search-based testing) compare to biasing selection towards: *purely technical code factors* such as 1) methods with more lines of code, 2) methods that are subjects of static analysis warnings, or *socio-technical factors* such as 3) methods involved in merge conflicts, 4) methods with bad code smells, 5) methods appearing in tests detecting faults in other changes by the same commit author. What factors determine which heuristics are most effective for a particular testing effort? Do the answers change when focusing on faults introduced by recent code changes vs. long-present low-probability faults? Answering these questions can help us move towards effective truly incremental software testing that works when integration is continuous and releases are challengingly frequent.

Synergy: The proposed project fits well in the context of our group's ongoing work on using *socio-technical* factors for fault prediction, and on improving heuristics for property-based and other time-constrained test generation methods.

References

[1] I. Ahmed, C. Brindescu, U. A. Mannan, C. Jensen, and A. Sarma. An empirical examination of the relationship between code smells and merge conflicts. In *International Symposium on Empirical Software Engineering and Measurement*, pages 58–67, 2017.

[2] I. Ahmed, R. Gopinath, C. Brindescu, A. Groce, and C. Jensen. Can testedness be effectively measured? In *International Symposium on Foundations of Software Engineering*, pages 547–558, 2016.

[3] I. Ahmed, C. Jensen, A. Groce, and P. E. McKenney. Applying mutation analysis on kernel test suites: an experience report. In *International Conference on Software Testing, Verification and Validation Workshops*, pages 110–115. IEEE, 2017.

[4] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. Mcminn, A. Bertolino, et al. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[5] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE transactions on software engineering*, 28(2):159–182, 2002.

[6] A. Groce, A. Fern, J. Pinto, T. Bauer, A. Alipour, M. Erwig, and C. Lopez. Lightweight automated testing with adaptation-based programming. In *International Symposium on Software Reliability Engineering*, pages 161–170, 2012.

[7] M. Harman, P. McMinn, J. T. De Souza, and S. Yoo. Search based software engineering: Techniques, taxonomy, tutorial. In *Empirical software engineering and verification*, pages 1–59. Springer, 2012.

[8] R. Just, D. Jalali, and M. D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *International Symposium on Software Testing and Analysis*, pages 437–440, 2014.

[9] G. J. Myers, C. Sandler, and T. Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[10] 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 (t). In *IEEE/ACM International Conference on Automated Software Engineering*, pages 201–211, 2015.

[11] P. Tonella. Evolutionary testing of classes. In *Software Engineering Notes*, volume 29, pages 119–128. ACM, 2004.

[12] J. Wegener and O. Bühler. Evaluation of different fitness functions for the evolutionary testing of an autonomous parking system. In *Genetic and Evolutionary Computation Conference*, pages 1400–1412. Springer, 2004.

[13] J. Wegener and M. Grochtmann. Verifying timing constraints of real-time systems by means of evolutionary testing. *Real-Time Systems*, 15(3):275–298, 1998.