Understanding and Mitigating Flaky Tests

Alex Groce (agroce@gmail.com)
Northern Arizona University

Google Contacts: James H. Andrews (jhandrews@google.com), Chaoqiang Zhang (chaoqiangzhang@google.com)
Google Sponsor: John Micco (jmicco@google.com)

1 Proposal

1.1 Abstract

This project proposes to investigate the causes of *flaky tests*, unreliable regression tests that can both pass and fail for the same code under test. The first phase of the project will gather more extensive, varied, and reliable empirical data on flaky tests, addressing weaknesses of previous limited investigations of flaky tests. By gathering a large set of tests (both flaky and non-flaky) from a large set of projects, flaky and non-flaky tests can be compared, and properties of likely-flaky tests identified. This analysis will also support the development and evaluation of a novel approach to mitigating flaky tests based on automatically decomposing large, complex tests into sets of simpler, less complex tests.

1.2 Problem Statement

In order to help ensure that they are reliable and secure, complex modern software systems usually include a large set of *regression tests*. A regression test suite is a (usually large) set of tests that can be run against a software system every time it is modified, to ensure that the modification has not broken the system in some way. *Flaky tests* [8] are software regression tests that fail in an intermittent, unreliable fashion, and thus degrade the utility of regression testing. The essence of a flaky test is that, for the same snapshot of test code and code under test (CUT), it sometimes fails and sometimes passes: the outcome of the test is not a deterministic property of the test code, code under test, and environment running the test. This produces three serious problems: first, a flaky test often wastes developer time by forcing the investigation of code-under-test that is not incorrect. Second, failures in flaky tests (for that reason) are often ignored, and therefore serious software faults may be missed. Finally, to mitigate these problems, flaky tests are often run multiple times, wasting valuable computing resources and delaying acceptance of code changes. A canary in a coal mine is of little use if canaries frequently become ill for reasons unrelated to the presence of toxic gases. Mining may stop for no good reason, or miners may learn to ignore the canary, leading to tragedy (a third, more "practical" option is that miners may carry so many redundant canaries into the coal mine that canary-care becomes a serious burden on mining).

In practice, the problem of flaky tests is not limited to regression testing. In our own work, we have encountered flaky tests produced by automated test generation systems, including production compiler fuzzers. The underlying problems are the same: flaky tests make software quality assurance more difficult and impose a large burden, in both computational and human terms. Better understanding of flaky tests, in terms of both underlying causes and statistical profiles of how they differ from other tests, is critical to direct present and future research on flaky tests. Novel approaches to mitigation for flaky tests, based on characteristics of flaky tests, rather than on the causes unique to each test, are also important to develop as a potential way to reduce the impact of flaky tests and the human effort needed to correct such tests.

1.3 Research Plan

Understanding Flaky Tests: Previous work on flaky tests has either focused on a single cause for flaky behavior (usually inter-dependence between tests [4]), or used only tests from one large open source project (Apache) [6, 9]. Of the latter studies, one [6] only used fixed flaky tests, making it impossible to compare

flaky tests to non-flaky tests in a statistical sense, and likely leading to a considerable bias in the analysis of underlying causes for flaky behavior. The other study [9] used a large body of tests, and detected flaky behavior at runtime, but did not produce a comprehensive comparison of tests and projects, instead focusing on the use of test smells. We propose to produce the first general, systematic examination of flaky tests, combining the merits of previous studies and overcoming their limitations. First, we will use a much larger body of tests, drawn from multiple large, mature open source projects; additionally we aim to use open-sourced large projects originally developed in an industry or government environment, in case open source development results in different test practices. We will augment these by generating additional tests using state-of-the-art automated test generation systems, in order to add understanding of a source of flaky tests not previously considered in the literature, and give a broader set of non-flaky tests for projects with limited test suites. Second, by using a large set of sampled tests (checked at runtime for flaky behavior) as well as fixes to identify flaky tests, we can contrast and compare flaky and non-flaky tests as well as (perhaps equally critically) flaky tests that are fixed and those that remain flaky. Finally, rather than only investigating root causes by manual analysis and examining a limited number of test smells, we will perform a wide variety of statistical comparisons between types of tests (non-flaky, flaky, flaky fixed, flaky not fixed, flaky before fix, flaky after fix) for numerous source code and executable attributes, e.g., source size and complexity, type of test (manual or automated, and if automated, from which tool), authorship, number of commits, entropy, language features used, deployment target (web server, stand-alone application, mobile app), and so forth. Rather than beginning with a hypothesis about flaky tests (as in the work on test smells [9]) this will allow us to discover surprising correlations with flaky behavior (or lack thereof).

Mitigating Flaky Tests: Better understanding of the attributes of flaky tests may, in itself, contribute to better handling of flaky behavior. Depending on the strengths of correlations, it may be possible to better tune systems for checking test failures for flaky behavior, or to warn when a developer produces a test that has high likelihood of being flaky. If there are no correlations strong enough to improve such measures, it will indicate a more pressing need for mitigations that go beyond detection to remedy flaky behavior.

Google engineer Jeff Listfield examined a large set of flaky tests at Google and discovered that being a flaky test correlated surprisingly well with sheer size of test executables [5], a finding we will attempt to reproduce over our data set of flaky tests drawn from non-Google projects. The relationship between flaky behavior and test size suggests that reducing the size of tests *could* be a way to reduce flaky behavior, but without better understanding how test code and CUT each contribute to executable size, and correlations for each factor measured independently (in terms of test lines of code and perhaps other complexity metrics) we cannot be sure. Better understanding of this relationship is one of the key aspects of our empirical study.

If test size is contributory to flaky behavior, it is unfortunate that manually reducing test size is difficult, time-consuming, and prone to mistakes. We propose to use our generalization of delta-debugging [10], *cause reduction* [3], to automatically produce multiple tests from a single large test, preserving code coverage and mutation detection, by working backwards from the last events (code coverage or mutation kills) in the sequence of testing. We hypothesize that by reducing instances where a single test has multiple independent events dependent on a single event or sequence, we can reduce the probability of flaky behavior. Smaller tests have fewer opportunities for concurrent interaction and resource use dependencies, for example. As a secondary benefit, we hope that decomposed tests will be easier to understand and use in debugging. Finally, even when one decomposition of a test is still flaky, in some cases only some of the tests produced may run due to a change, reducing the chance of executing a flaky test.

Synergy: The proposed project fits well in the context our our group's ongoing DARPA BRASS project (`https://www.darpa.mil/program/building-resource-adaptive-software-systems`) funded work on using source-code reduction to synthesize resource adaptations for systems, using tests as specifications of variant behavior [1]. Developing effective tools for reducing code while retaining semantic properties is a key feature of both the DARPA work and our proposed mitigation method. More generally, analysis of large source code repositories and investigation of test properties is the primary thrust of our research group, over years of NSF and DARPA funded work, and general manipulation of test code is the topic of an NSF proposal we will be submitting in November.

1.4 Prior Work

Gao et al. and others have generally considered the question of how to make tests repeatable [2]. The most extensive previous study of flaky tests in practice was that of Luo et. al [6]. Listfield analyzed actual Google tests [5], and found executable size a likely contributing factor. Palomba and Zaidman [9] investigated the relationship between test code smells and flaky tests. There is also considerable work on particular causes for a test being flaky, such as dependence on ordering of tests [4], but no work we know of on general mitigation of flaky behavior across causes. How to handle flaky tests in practice (when they cannot be avoided) is a major issue in Google-scale continuous testing [7], and, as Memon et al. describe, the problem of flaky tests influences the general design of Google test automation.

References

[1] A. Christi, A. Groce, and R. Gopinath. Resource adaptation via test-based software minimization. In *IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, page accepted for publication, 2017.

[2] Z. Gao, Y. Liang, M. B. Cohen, A. M. Memon, and Z. Wang. Making system user interactive tests repeatable: When and what should we control? In *International Conference on Software Engineering*, ICSE '15, pages 55–65. IEEE, 2015.

[3] A. Groce, M. A. Alipour, C. Zhang, Y. Chen, and J. Regehr. Cause reduction: Delta-debugging, even without bugs. *Journal of Software Testing, Verification, and Reliability*, 26(1):40–68, 2016.

[4] W. Lam, S. Zhang, and M. D. Ernst. When tests collide: Evaluating and coping with the impact of test dependence. Technical Report UW-CSE-15-03-01, University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, Mar. 2015.

[5] J. Listfield. Where do our flaky tests come from? `https://testing.googleblog.com/2017/04/where-do-our-flaky-tests-come-from.html`, April 2017.

[6] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. An empirical analysis of flaky tests. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 643–653. ACM, 2014.

[7] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco. Taming Google-scale continuous testing. In *International Conference on Software Engineering*, pages 233–242. IEEE, 2017.

[8] J. Micco. Flaky tests at Google and how we mitigate them. `https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html`, May 2016.

[9] F. Palomba and A. Zaidman. Does refactoring of test smells induce fixing flaky tests? In *IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2017.

[10] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

2 Data Policy

All analysis data from this project based on open source projects will be made public, along with analysis scripts, via GitHub or similar open source hosting solution. Tools for mitigation will be hosted in the same way. Depending on size of artifacts, only pointers to the actual repository artifact snapshots analyzed may be posted, rather than their full contents (since we expect to potentially analyze very large amounts of source code that is already accessible online).

3 Budget

The budget of $49,275 will support one PhD student for one year, including travel:

- Stipend/summer $22,100 + Fringe Benefits $2,436 + Tuition $9,700 = Student Support: $34,236
- Conference travel: $1,500
- Overhead (52%) on non-tuition items: $13,539