ABSTRACT

Title of dissertation: QUANTIFYING FLAKINESS AND MINIMIZING ITS EFFECTS ON SOFTWARE TESTING

Zebao Gao
Doctor of Philosophy, 2017

Dissertation directed by: Professor Atif Memon
Department of Computer Science

In software testing, test inputs are passed into a system under test (SUT); the SUT is executed; and a *test oracle* checks the outputs against expected values. There are cases when the same test case is executed on the same code of the SUT multiple times, and it passes or fails during different runs. This is the *test flakiness problem* and such test cases are called *flaky tests*.

The test flakiness problem makes test results and testing techniques unreliable. Flaky tests may be mistakingly labeled as failed, and this will increase not only the number of reported bugs testers need to check, but also the chance to miss real faults. The test flakiness problem is gaining more attention in modern software testing practice where complex interactions are involved in test execution, and this raises several new challenges: What metrics should be used to measure the flakiness of a test case? What are the factors that cause or impact flakiness? And how can the effects of flakiness be reduced or minimized?

This research develops a systematic approach to quantitively analyze and min-

imize the effects of flakiness. This research makes three major contributions. First, a novel *entropy-based metric* is introduced to quantify the flakiness of different layers of test outputs (such as code coverage, invariants, and GUI state). Second, the impact of a common set of factors on test results in system interactive testing is examined. Last, a new *flake filter* is introduced to minimize the impact of flakiness by filtering out flaky tests (and test assertions) while retaining bug-revealing ones.

Two empirical studies on five open source applications evaluate the new entropy measure, study the causes of flakiness, and evaluate the usefulness of the flake filter. In particular, the first study empirically analyzes the impact of factors including the system platform, Java version, application initial state and tool harness configurations. The results show a large impact on SUTs when these factors were uncontrolled, with as many as 184 lines of code coverage differing between runs of the same test cases, and up to 96% false positives with respect to fault detection. The second study evaluates the effectiveness of the flake filter on the SUTs' real faults. The results show that 3.83% of flaky assertions can impact 88.59% of test cases, and it is possible to automatically obtain a flake filter that, in some cases, completely eliminates flakiness without comprising fault-detection ability.

QUANTIFYING FLAKINESS AND MINIMIZING ITS EFFECTS ON SOFTWARE TESTING

by

Zebao Gao

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2017

Advisory Committee:
Professor Atif Memon, Chair/Advisor
Professor Ashok Agrawala
Professor James Purtilo
Professor Gang Qu
Professor Alan Sussman

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

1.1 Testing and Test Oracles

In software testing, test inputs are given to a system under test (SUT) and *test oracles* are used to determine whether the test passed or failed by comparing the observed outputs against expected outputs often encoded in assertions. For example, to test a *grep* [2] function, the inputs may include the text and the pattern to search, and the oracle can assert certain values for search results. In unit testing, the inputs are typically supplied as parameters to methods, and test results are checked by assertions on certain variables, return values, or other outputs. Figure 1.1 shows an example JUnit[1] test case. The upper frame shows the program under test, which includes a method named *reverseDigits()* that takes a number as input and returns a number with all digits reversed as output. As shown in the lower frame, a JUnit test case named *testPositiveNumber()* passes in a value 501, and asserts a return value 105. The test will pass if and only if the return value of the method is 105.

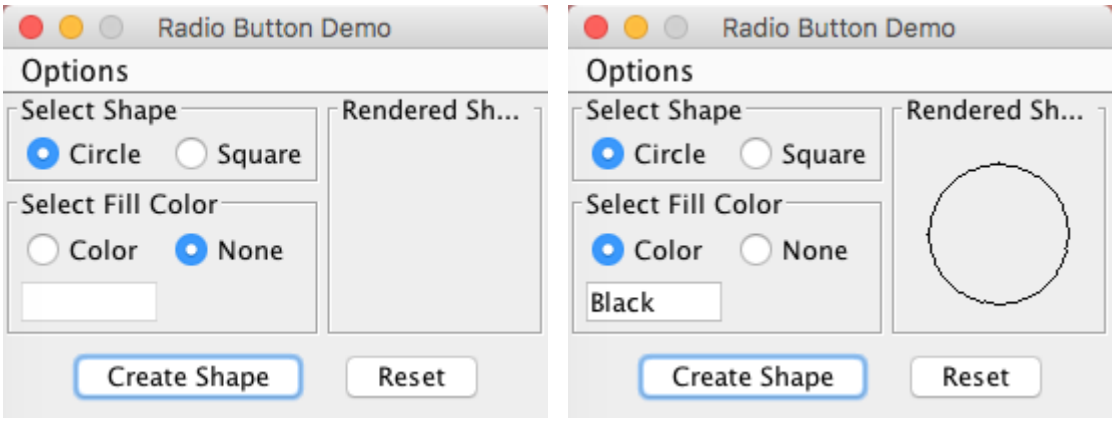
System User Interactive Tests (SUITs), on the other hand, involve more complex inputs and outputs. For example, in a remote procedure call (RPC) system, asynchronous or blocking remote calls are made, and the returning messages need

[1] http://junit.org/junit5/

```java
// A method under test that takes in a number and returns a number
   with all digits reversed.
public class ReverseDigits {
   public static int reverseDigits(int n) {
      int reversed = 0;
      while (n != 0) {
         reversed = reversed * 10 + n \% 10;
         n = n / 10;
      }
   }
}
```

```java
// A JUnit test for the reverseDigits() method.
import org.junit.test;
import static org.junit.assert.assertions;
import static ReverseDigits.reverseDigits;
public class ReverseDigitsTest {
   @Test
   public void testPositiveNumber() {
      assertEquals(reverseDigits(501), 105);
   }
}
```

Figure 1.1: Example JUnit Test Case



(a) Before Test Actions (b) After Test Actions

Figure 1.2: Interface of SUT Before and After Test Actions

to be decoded and then validated. As a concrete example, an *AJAX* [3] web client may send a RPC request to the server, parse the returned *Json*[2] object, and update its display or internal state upon receiving the response from the server. Further, when testing an application with a graphical user interface (GUI), inputs may come from system events on GUI widgets (e.g., clicking on a button, expanding a menu, etc.), and assertions may check various properties of the application's active widgets. For example, to test a GUI-based text editor, the test case may drive the application to perform copy-paste commands, and check if the text content of the editor panel matches expectations. As a concrete example, Figure 1.2 shows the GUI of an application that allows users to draw a selected shape in a selected color. The screenshot in Figure 1.2(a) shows the original interface of the application. A SUIT created using the HP Unified Functional Testing (UFT, previously known as Quick Test Professional, or QTP)[3] is shown in Figure 1.3. The first part of the

[2]http://json.org/example.html

[3]https://software.microfocus.com/en-us/software/uft

tests are actions performed on the UI, such as setting the shape, selecting the color, and clicking on the "Create Shape" button. The UI after these actions is shown in Figure 1.2(b). In the test script, several assertions are made on the new interface to validate that the correct shape is rendered, and that properties such as "text", "foreground (color)", and "width" of the rendered shape have expected values. The test will fail if the expected Circle Panel is not shown, or any of the property values are not as expected.

1.2 Context for This Work

Testing is done at multiple points in the software development process. One important point is the Continuous Integration (CI) cycle, in which test cases act as gatekeepers for a new software modification. Because the CI cycles are extremely tight (every few hours), it is important that the test cases be reliable. The context of this research is in the CI cycle. There is a place for additional test cases that may provide useful information to a developer or tests, but may be unreliable, e.g., due to concurrency. Such test cases not desired in the CI cycle and thus are outside the scope of this work.

1.3 The Flakiness Problem

Numerous development and maintenance tasks require the automated execution and re-execution of test cases, and often these are run from the system or user interface perspective [4]. For instance, when performing regression testing on a web

```
// Test actions.
JavaWindow("Radio Button Demo").JavaRadioButton("Circle").Set
JavaWindow("Radio Button Demo").JavaMenu("Options")
          .JavaMenu("Color").JavaMenu("Black").Select
JavaWindow("Radio Button Demo").JavaButton("Create Shape").Click


// Assert a Circle Panel is shown.
if Not JavaWindow("Radio Button
   Demo").JavaObject("CirclePanel").Exist(1) Then
   Reporter.ReportEvent micPass, "micFail",
         "The Circle Panel SHOULD show up"
end if


// Check properties of the Circle Panel.
JavaWindow("Radio Button Demo").JavaObject("CirclePanel")
          .CheckProperty "text", "Rendered Shape"
JavaWindow("Radio Button Demo").JavaObject("CirclePanel")
          .CheckProperty "foreground", "333333 Shape"
JavaWindow("Radio Button Demo").JavaObject("CirclePanel")
          .CheckProperty "width", "112"
```

Figure 1.3: Example SUIT Created Using HP UFT

application [5] or integrating components with a GUI, a test harness such as Selenium [6], Mozmill [7], or Unified Functional Test (UFT) [8] may be used to simulate system-level user interactions on the application by opening and closing windows, clicking buttons, and entering text.

Over the past years, numerous researchers have proposed improved automated testing and debugging techniques [3, 9–17]. Common metrics have been used to determine success such as fault detection [13,18,19], time to early fault detection [20], business rule coverage [16], and statement, branch and other structural coverage criteria [21]. Test assertions for these applications operate at differing degrees of precision [22] such as detecting changes in the interface properties [23], comparing outputs of specific functions [24], or through the observation of a system crash [25].

The assumption for all of this previous work, however, is that these tests can be reliably executed or replayed, i.e., they produce the same output (code coverage, invariants, object states) unless either the tests or SUT changes. In our own experience with testing and benchmarking [26–28], we have learned that it is hard to repeat test results with different platforms and configurations. Specifically, we found that even within the same platform, simple changes to system load or a new version of execution platform, or even the time of day that we ran a test, could impact code coverage or the application interface state. The phenomenon that the same tests on the same SUT may behave differently in different runs is referred to as the *test flakiness problem*, and these tests are often called *flaky tests* [28]. The flakiness problem is gaining much attention in recent years in industry practice – several testing frameworks now support annotations for flaky tests [29] [30].

Flaky tests are gaining attention in industry practice due to the problems they create and resources they demand. In the most common scenario of automatic testing, flaky tests tend to be reported as failed, which either needs re-execution – meaning more computation resources – or manual effort such as checking bug reports. At Google, a failed test case will be re-run on the same code 10 times, and if the test passed in any subsequent run, then it is labeled as flaky instead of failed [31]. According to a recent technical report, 16% of their 3.5 million tests have some level of flakiness, and 84% of transitions of tests from passing to failing are due to flakes. Consequently, Google is spending up to 16% of their CI compute resources re-running flaky tests. Taking into consideration the size of Google's compute resource pool, flaky tests consume significant resources in practice. Other companies such as Netflix [32] and VMWare [33] also report problems with flaky tests.

Recent studies provide different perspectives on the flakiness problem, however, none agree on how to define flakiness. Memon and Cohen [28] proposed in a tutorial that when test execution environments are uncontrolled, tests may become flaky, i.e., may generate different results. The tutorial did not formally define flaky tests or specify factors of the execution environment to be controlled. Recent work by Luo et al. [34] empirically studied causes for reported flaky tests from the Subversion repository of the Apache Software Foundation[4]. They also did not include a formal definition for flaky tests. Instead, they studied tests reported by the developers or testers as "flaky" in the Apache Repository: the tests *may* be reported

[4]http://svn.apache.org/repos/asf/

flaky due to inconsistent results generated by executing the test suite in the same or different orders, on the same or changed code. Google, in contrast, labels a test as flaky if it passed in some runs but failed in others on the same code. Without a formal definition of flakiness, we cannot measure, reason about, or minimize its effects.

1.4 Thesis Statement

This research aims to tackle the aforementioned problems associated with test flakiness. This leads to our thesis statement:

We hypothesize flakiness of tests can be quantified, the underlying factors of flakiness can be identified, and effects of flakiness can be minimized.

1.5 Challenges and Contributions

This research develops a systematic approach to tackle the major challenges raised by the flakiness problem.

Challenge 1: It is challenging to measure flakiness.

Contribution 1: This research develops a novel metric to quantify the flakiness of a test or test suite.

Each time a test is executed, the test outputs can be examined at different layers. More specifically, this research studies: (1) *User Interaction Layer:* this layer is typically used to execute SUITs and extract GUI widget properties for test oracles, (2) *Behavioral Layer:* to infer properties, such as invariants, regarding the

test execution, and (3) *Code Layer*: for code coverage.

This research formally defines a flaky test or test suite as:

Definition 1 (Flaky Test (Suite)) *A test (suite) is* flaky *in multiple runs at a certain output layer* iff *the outputs at the layer are not identical across runs on the same SUT with the same configuration.*

Based on this definition, for each specific SUT, we first executed a fixed test or test suite multiple times and captured outputs at a certain output layer. Then we developed an entropy-based metric to measure the flakiness of a test based on probabilities of observing its different outputs. A flaky test may have various outputs across different runs, and consequently, the entropy of the test's output serves as an indicator of its flakiness. For instance, all the runs may have the same output, and our entropy-based metric will give an entropy value of 0, meaning the test is not flaky. As another extreme case, all runs may have different outputs and our metric will give the maximum possible entropy value, showing the test is very flaky. All other tests will have an entropy value in between the two extremes, depending on the structure of the outputs.

This entropy-based metric lays the foundation for all further work: it is utilized to quantify test flakiness when studying the impacting factors; and utilized to minimize the effects of flakiness by filtering out flaky failures.

Challenge 2: It is challenging to understand causes of flakiness.

Contribution 2: This research identifies a common set of factors that may impact flakiness and evaluate their impact at different output layers.

Identifying factors that may impact test flakiness and analyzing their impacts is an important task, because it helps us understand the underlying nature of the problem and provides clues on how to control flakiness. The research begins with identifying key factors including (1) test execution platform, (2) application starting state or configuration, (3) test harness factors, and (4) execution application versions. Then we set up a set of configurations to cover various operating systems, program initial state, time delay between steps, and Java Runtime Environment (JRE) versions, and so on. For each configuration, the same test suite was executed on each SUT 10 times and the test outputs were observed at the three output layers. Finally the entropy-based metric was used to calculate the flakiness of each test and test suite. This research studies the impact of each factor by comparing flakiness of tests of different SUTs under different configurations.

Results on 5 open-source Java GUI programs of various sizes show that the impact is large for many applications when factors (including initial state, step delay, and so on) are uncontrolled – as many as 184 lines of code coverage differ and more than 95% false positives for fault detection were observed.

Despite our ability to control factors and reduce variance, there still may exist some that we cannot control - that are application specific, or sensitive to minor changes in timing or system load - therefore a single run of any testing technique (despite fault determinism) may lead to different code coverage or even different invariants, and that unless one accounts for the normal variance of a subject, a single test run may be insufficient to claim that one technique is better than another, or even that a fault is really a true fault.

Challenge 3: It is challenging to minimize the effects of flakiness.

Contribution 3: This research develops a *flake filter* to de-flake a test suite without impacting its fault detection ability.

In a typical regression testing scenario, test cases including assertions are created and executed on the original version; and then executed on subsequent versions to detect regression bugs. Failures reported on the subsequent versions may fall into 3 categories: (1) *true failures* caused by bugs, (2) *flaky failures* due to flakiness, and (3) *update failures* caused by feature changes. Flaky tests and assertions increase the amount of failures testers or developers have to go through to identify non-flaky failures that reveal bugs (true failures) or obsolete tests (update failures).

To minimize the effects of flakiness, this research develops a flake filter that automatically filters out flaky tests and assertions. We utilize our entropy-based metric to measure the flakiness score of a test case or assertion based on its results over multiple runs on a single version of the SUT. And if the score exceeds a manually specified *threshold*, the test or assertion is no longer used to detect failures in the subsequent version. By tuning the threshold, the filter can achieve best overall performance in eliminating flaky tests and retaining fault-detection ability of tests.

A different set of SUTs together with 16 real bugs reported on their bug reporting sites were used for our study. Results show that a small percentage of flaky assertions (1.03%-7.19%) can result in very large percentages of flaky test cases (54%-100%). The flake filter is able to significantly reduce this impact. For some bugs, the flake filter is able to completely de-flake the tests without impacting their fault detection ability.

To summarize, this research has 3 goals: measuring, understanding the causes of, and minimizing the effects of flakiness. For the first goal, the research develops an entropy-based metric to quantify flakiness. For the second goal, the approach is to identify a set of factors and to empirically study their effects on test flakiness. And for the third goal, the research develops a flake filter that eliminates flaky tests and assertions in regression testing.

1.6 Contributions and Disseminations

This section itemizes specific contributions (labeled as C1, C2, and so on) of this research and summaries resulting research papers published at various venues.

C1 Formalizing the definition of test flakiness.

C2 Developing an entropy-based metric to quantify flakiness.

C3 Measuring test flakiness at three output layers: the code layer, the invariant layer and the user-interface layer.

C4 Identifying factors that impact flakiness of SUITs and performing a comprehensive experiment to study the impacts of factor.

Results related to above contributions were published at the 2015 IEEE 37th International Conference on Software Engineering (acceptance rate = 18.5%) [35]. Typical metrics for determining effectiveness of various testing technique include code coverage and fault detection, with the assumption that there is determinism in the resulting outputs. We developed an entropy-based metric to quantitatively

examine the extent to which a common set of factors such as the system platform, Java version, application starting state and tool harness configurations impact these metrics. We also examined three layers of testing outputs: the code layer, the behavioral (or invariant) layer and the external (or user interaction) layer.

C5 Performing a set of feasibility studies on minimizing effects of flakiness by eliminating flaky test cases and assertions.

We first studied a technique that eliminated failures related to object properties that have been observed "unstable", thereby reducing the effect of flakiness. The results were published at 2015 IEEE 26th International Symposium on Software Reliability Engineering (acceptance rate = 32.0%) [36].

In addition, a preliminary study on a small number of bugs and test cases percolated real failures to the top of a set of reported failures using an entropy-based metric. The results were published at the TestBEDS Workshop at 2015 IEEE/ACM 30th International Conference on Automated Software Engineering [37].

C6 Developing an automatic flake filter that automatically eliminates flaky failures.

C7 Conducting an experiment on subject applications with real faults to evaluate the effectiveness of the flake filter.

The previously reported results relied on human expertise to manually identify a universal set of "unstable" object properties for an SUT. We further developed a fully automatic approach to reduce the effects of flaky tests. By measuring the

entropy of each object being retrieved by each assertion, we developed a flake filter to automatically weed out flaky failures. We evaluated our technique on 16 real bugs on 18 different versions of 3 open source Java GUI applications. And this work has been submitted to 2018 IEEE 11th International Conference on Software Testing, Verification and Validation [38].

The research work discussed next did not contribute directly to the main thrust of this thesis, but served to inform about existing challenges in software testing, especially the test flakiness problem; it has also been published at various venues.

In regression testing, whenever the GUI changes – widgets get moved around, windows get merged – some scripts become unusable because they no longer encode valid input sequences. Moreover, because the software's output may have changed, their test oracles – assertions and checkpoints – encoded in the scripts may no longer correctly check the intended GUI objects. To address these challenges, we developed ScrIpT repAireR (SITAR), a technique to automatically repair unusable low-level test scripts. This was published in 2016 IEEE Transactions on Software Engineering [39].

In another study, we developed a prototype tool named VGT GUITAR that utilizes Computer Vision techniques to test GUIs; and evaluated its advantages and disadvantages when compared to traditional object-based testing techniques. This work was published at 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (acceptance rate = 25%) [40].

Mutation has been widely used to measure fault-detection ability of various techniques. We developed GUI mutation operators at the Mutation Workshop at 2015 IEEE 8th International Conference on Software Testing, Verification and Validation [41].

Finally, continuous testing and integration is a challenging problem at Google due to its scale of code and frequency of code repository updates. Even with enormous resources dedicated to testing, regression test each code change individually is unrealistic, resulting in increased lag time between code check-ins and test result feedback to developers. We reported results of a project that aims to reduce this time by: (1) controlling test workload without compromising quality, and (2) distilling test results data to inform developers, while they write code, of the impact of their latest changes on quality. This work was published at 2017 IEEE 39th International Conference on Software Engineering, Industry Track (acceptance rate = 29%) [42].

1.7 Broader Impacts

This research develops a systematic approach towards measuring and minimizing the effects of test flakiness, and has the potential to broadly impact software testing practice and research.

Test flakiness is not an isolated problem. It has potential impacts on most functional testing techniques. For example, fault localization techniques need to differ between the passed and failed tests; regression testing needs to compare the

test results between two versions to determine broken features, or to prioritize tests; mutation testing compares the results of the original and mutated versions to determine whether tests can kill mutants. All of these techniques, if developed and evaluated upon unreliable and flaky tests, will yield incorrect results.

The next chapter introduces the background of the work. The subsequent 3 chapters present approaches addressing the 3 challenges and supporting experimental results. The final chapter summarizes our conclusions and possible future research directions.

Chapter 2: Background

This chapter elaborates on existing work related to the current research effort. Two major topics are covered in this chapter: previous studies related to test repeatability and flakiness, and techniques utilized in this research.

2.1 Studies on Test Repeatability

A number of research efforts have studied the test repeatability problem, but most have focused on the impact of a single factor on test execution.

2.1.1 Test Dependence

A test is dependent in a test suite *iff* there exist at least two orderings of the test suite that cause different outputs. Many techniques (implicitly) assume test independence. For example, many test prioritization [43] and selection [44] techniques used to find bugs more quickly use an objective function to assign priorities or select tests. The most commonly used objective function is the code coverage of the test cases. Such techniques typically assume the coverage for each test remains the same in different permutations, and therefore, that the tests are independent.

Zhang et al. empirically studied the test independence assumption [45]. They

formally defined test dependence and developed algorithms that detect dependent tests from human-written and automatically-generated test suites in their studied subjects. While the test independence assumption is often implicitly made, some researchers have taken this issue into consideration when developing their testing techniques [46–48].

It is important to note that test dependence is different from determinism: independent tests, even though not affecting one other, may contain nondeterministic code. On the other hand, dependent tests may not contain nondeterministic code but always affect each other. Most of the previous studies on test dependency neglect the fact that executing the same sequence of tests on exactly the same SUT could generate different test results. In this research, we eliminate the impact of test dependency by always fixing test execution order, and then further studying the nondeterminism inside the test.

2.1.2 Concurrency and Timing Issues

Concurrent programs are prevalent nowadays due to the ubiquity of multi-core processors on servers and desktops. GUI applications are among the most common examples of concurrent programs where multiple threads take charge of GUI rendering and backend computing. However, it is difficult to test concurrent systems due to their non-determinism: (1) some bug revealing execution traces may be rarely covered in tests; and (2) it is difficult to reproduce previous bug-revealing traces for debugging.

Farchi et al. categorized a taxonomy of concurrent bug patterns together with heuristics to help detect these bugs [49]. Lu et al. studied 105 real-world concurrency bugs from representative open source applications and examined the bug patterns, manifestations and fix strategies [50].

Automatically identifying bugs associated with unexpected interleaving of threads is even more challenging. Stoller et al. presented a randomized scheduling technique which inserts scheduling function calls at concurrent events [51]. Hartman et al. proposed a more systematic but less scalable model-based test-generation technique for validation of parallel and concurrent software [52]. Yan et al. presented a method of test generation based on BPEL, a language that could express concurrent behaviors of software [53]. And Pugh et al. developed a framework to construct unit tests to test block/unblock behaviors in multithreaded programs [54]. Despite the research efforts, none of them focus on rare probabilistic faults that can exist in both hand-coded and automated-generated tests.

Most of the automatic test generation techniques specifically designed for concurrent software focus on unit level tests. Even through it has never been demonstrated, researchers believe that system tests suffer from severe timing problems: system tests often involve more complex computations and interactions not only between threads but also human and computers. GUI testing frameworks like Selenium [55] and GUITAR [56] provide syncing mechanisms like *wait()* or *waitFor()*, which wait for a specific time or event to occur such as the rendering of a GUI widget. There still lack effective techniques to automatically determine how long a test needs to wait and which elements to wait for. GUITAR uses a heuristic that

periodically checks if the system event queue is empty, while this heuristic works well in practice, it cannot always ensure the handler of a previous event has successfully finished before moving on to the next event. In fact, most previous system level testing techniques [1] assume determinism in test results. This research studies the flakiness problem in system level tests.

2.1.3 Other Factors

Arcuri et al. [57] developed a method to generate unit tests that behave differently in varying environments, controlling this by replacing API calls and classes related to the environment with mocked versions. Although the test execution can become more stable based on mocked dependencies, the major goal of their technique is to increase the code coverage of the developed classes, instead of the repeatability of testing the original system by controlling the real environment. In addition, there are other studies regarding other factors such as the initial state [58], system loads [59] and network connection [60, 61].

2.2 Previous Studies on Test Flakiness

The problem of flaky tests first attracted attention of industry practitioners [31]. Some test frameworks provide annotations for flaky tests [29, 30]. At Google, flaky tests are considered one of the leading cause of inefficiency: 2-16% of the computational resources of continuous integration are spent re-executing flaky tests [62]. To the best of our knowledge, Memon et al. firstly presented the test flaki-

ness problem to the research community [28] by suggesting controlling factors in test execution to avoid flakiness. In addition, Luo et al. studied the flaky tests reported in Apache projects and categorized the major reasons and fixes for them [34].

Despite its recent attention, there is currently no systematic approach to the flakiness problem. First, as mentioned in the previous chapter, there is no universal definition for flaky tests. Luo et al. studied the "flaky tests" reported by developers of the subject applications. The paper mentioned that the tests *may* be caused by various test execution factors, such as the order of test execution and even change of code in the SUT. Luo et al. fail to provide a clear definition of flaky tests, however, and no metric is proposed to measure flakiness. Second, the factors that impact flakiness need to be further studied. Memon et al. mentioned that some factors need to be controlled for automatic testing, but did not specify which factors and how they will impact flakiness [28]. Luo et al. studied 161 reported flaky tests and classified their causes into groups like concurrency and test order dependency [34]. However, it remains unclear how some test execution factors (like test execution platform, initial state of subject of application, timing, etc) can affect flakiness. This research proposes to quantitatively study how these factors can impact flakiness. The results will provide testers with guidelines to minimize test flakiness in practice. Last, even though some general guidelines were given in previous work, the research community has not taken flakiness into consideration when designing test oracles. This research proposes to develop test oracles that minimize flakiness.

2.3 Test Execution

Now we introduce the tools we use to generate and execute our GUI tests. The main experiments use the GUITAR framework to test Java GUI applications. Since the test harness itself can be a factor that impacts the test flakiness, the research proposes to conduct a complementary experiment where another widely used tool, *Selenium*, will be used to drive the execution of web applications.

2.3.1 The GUITAR Framework

We start with a description of GUITAR, our automatic GUI Testing frAmeworRk. GUITAR automatically builds a formal model called an Event-Flow Graph (EFG) of the application under test (SUT), generates test cases based on an event coverage criteria, and replays them.

GUITAR adopts a process called GUI Ripping [63] to traverse the available events on the GUI in a depth-first manner. During this process, the GUI ripper extracts GUI structure information, including the hierarchical structure of GUI windows and widgets, as well as their properties (e.g., title, type, position, whether a widget opens a modal/modeless *Restricted-focus events* open *modal windows*, *unrestricted-focus events* open *modeless windows*, and *termination events* close modal windows.

The GUI Ripper then converts the GUI structure to an EFG, which is a directed graph representing all possible event interactions in the GUI. More formally, an *EFG* for a GUI G is a 4-tuple $<\mathbf{V}, \mathbf{E}, \mathbf{B}, \mathbf{I}>$ where: (1) $\mathbf{V}$ is a set of vertices

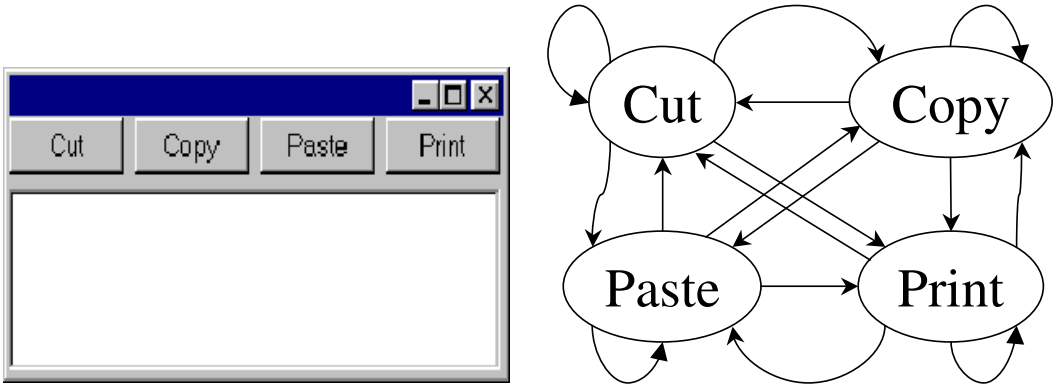


Figure 2.1: Example GUI and its EFG

representing all the events in G. Each $v \in \mathbf{V}$ represents an event in G; (2) $\mathbf{E} \subseteq \mathbf{V} \times \mathbf{V}$ is a set of directed edges between vertices. We say that event e_i **may-follow** e_j iff e_j may be performed *immediately* after e_i. An edge $(v_x, v_y) \in \mathbf{E}$ iff the event represented by v_y **may-follow** the event represented by v_x; (3) $\mathbf{B} \subseteq \mathbf{V}$ is a set of vertices representing those events of G that are available to the user when the GUI is first invoked; and (4) $\mathbf{I} \subseteq \mathbf{V}$ is the set of restricted-focus events of the GUI.

That is, in an EFG, each vertex represents an event on the GUI (e.g., click-on-File, click-on-Open), and an edge represents a *may-follow* relationship between two events. Note that only events inside the modal window invoked will follow a restricted-focus event. All events in the new invoked window as well as the original invoking window will follow an unrestricted-focus event. All events in the window from which the current modal window is invoked will follow a termination event. In the example shown in Figure 2.1, a *may-follow* edge from event *Copy* to another event *Paste* means the latter event *may* be performed immediately after the former event.

Each test case generated is a sequence of events from the EFG. More formally, a *test case* is $<e_1, e_2, e_3, \ldots, e_n>$ where $(e_i, e_{i+1}) \in \mathbf{E}$, $1 \leq i \leq n-1$. Notice that each test case will need to start from an event available at the initial state of the SUT, thus *reaching events* may be prepended to the test case to make it executable. Event coverage criteria are used to generate GUI test cases. For example, a test suite, T_E, can be generated to cover all events in the EFG. Consider a test suite T = $\{t_1 = <Copy>, t_2 = <Cut, Paste>, t_3 = <Cut, Print>\}$ that covers all events in the GUI. A test suite, T_D, can also be generated in a similar manner to cover all edges in the EFG. GUITAR executes the test cases one by one from the same initial state of the application, capturing the GUI's state after each event. This state will be used to create the *test oracle*, a mechanism used to determine whether a test case passed or failed.

2.3.2 Selenium

Selenium is a widely used framework "for automating web applications for testing purposes" [1]. The framework includes two key components: the *Selenium WebDriver* which provides an API to drive web browsers using each browser's native support for automation; and the *Selenium IDE* which is a complete integrated development environment that enables record-and-replay of user interactions with a web application as well as script editing.

In this research, web tests will be generated and executed using the Selenium IDE which allows testers to record, edit and replay Selenium scripts. **Record:** The

[1] http://www.seleniumhq.org

testers can interact with the web application by performing actions in a test case, like clicking on a link or typing in text, etc. The IDE will automatically capture all the actions together with the parameters and record them as script lines. **Edit:** The testers can modify the scripts as needed. For example, some input values can be replaced with variable names to make the scripts more generally usable. Also, the testers may insert *assert* statements to check a certain property of a certain web element (e.g., the title of a popup dialog) and the script will quit execution when the assertion fails; or a *waitFor* statement can be inserted which makes the script wait until a certain element shows up or a certain statement turns true. **Replay:** Finally, the recorded and edited scripts can be automated replayed (in regression testing). The test passes if the script finishes execution and all assertions are true and fails otherwise.

2.4 Test Output Observation

2.4.1 The Daikon Invariant Detector

A tool named *Daikon* is used to dynamically detect likely program invariants in this research. According to Ernst [64], "A program invariant is a property that is true at a particular program point or points, such as might be found in an *assert* statement, a formal specification, or a representation invariant." And the following are some examples of invariants: $y = 2 * x + 1$; $x >= y$; $size(keys) = size(entries)$; and *graph g is acyclic*. Invariants are helpful in tasks from software design to maintenance. For example, some invariants must be preserved when the code is

modified. However, invariants are often missing from programs.

The *Daikon* tool requires three major steps to obtain invariants: target program instrumentation, test execution, and invariants inference. **Instrumentation:** The Daikon invariant detector is a dynamic analysis technique which discovers likely program invariants from program execution data. Thus the tool needs to first instrument the target program to trace runtime values of certain variables. **Test Execution:** Daikon relies on the test suite of the target program which can be executed on the instrumented program to collect runtime variable values for analysis. Thus the quality of the inferred invariants inevitably depends on the quality and comprehensiveness of the test suite. In practice, "standard" test suites that is large enough and with reasonably good coverage have performed adequately in invariant detection. However, the technique cannot guarantee the completeness or soundness of its results. Also, Daikon assumes a distribution and performs a statistical test over the observed values of each variable to eliminate values generated by chance. **Invariants Inference:** Daikon developed a list of invariants that will be checked, including unary invariants involving a single variable (such as $x = a$), binary invariants over two variables (such as $x \in arr$) and ternary invariants over a fixed set of operands. Daikon also checks derived invariants within a certain depth of derivation from derived variables. The derived invariants for a numeric sequence, for example, can be the *length* or *sum* of the sequence. Daikon will test all the possible invariants over the trace data to obtain likely invariants.

2.4.2 GUI Test Oracle

We use the GUI state for the test oracle because any part of the GUI state through the sequence of the test execution may be bug revealing. Figure 2.2 shows an example of using GUI state for test oracle. The test case is a sequence of actions with the "Cancel" action at the ith position. The GUI state S_i shows the GUI state after this step. The GUI state consists a list of triples consisting of the *widget identifier* (e.g., "Cancel"), the *property name* (e.g., "Color" and "Height") and the *property value* (e.g., "Grey" and "40"). The GUI state includes all available GUI properties of all widgets in all shown windows of the SUT at this step. For the purpose of test oracle, the captured GUI state can be compared against the expected GUI state to determine if all the GUI widget properties are as expected.

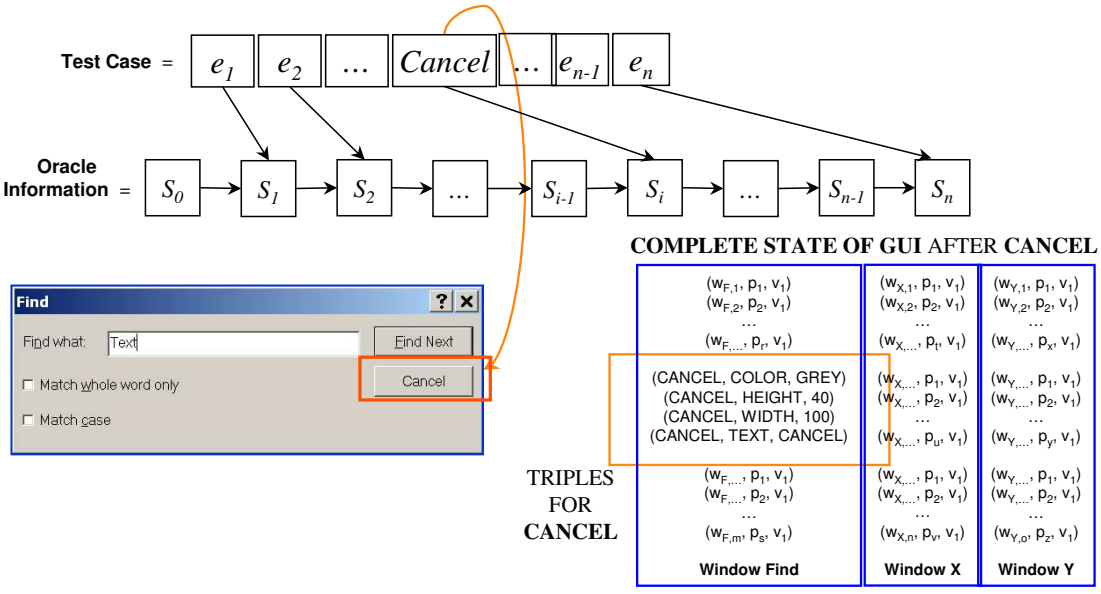


Figure 2.2: Capture GUI State For Test Oracle [1]

Xie and Memon presented automatic test oracles on applications with manu-

ally seeded bugs [1]. More importantly, they raised two important research questions regarding GUI test oracle: (1) what to assert, and (2) how frequently to check an assertion. They proposed 3 options regarding what to assert: (1) properties of the GUI widgets associated with the event, (2) properties of all GUI widgets in the current active window, and (3) properties of all GUI widgets of all windows. In addition, 2 options are suggested regarding how frequently to check an assertion: (1) after execution of each event, and (2) after the last event of the test case. This results in 6 different combinations of GUI test oracles. Their study shows that comparing all properties at the end of test case execution provides the most effective test oracle.

In this research, we further improve the test oracles by automatically filtering out failures due to flaky object properties.

Chapter 3: Quantifying Flakiness

To quantify test flakiness, this research develops an entropy-based metric to measure the flakiness of a test or test suite. Because the flakiness is measured based on test outputs on a certain output layer, we will first present the different output layers then the metric to measure flakiness will be introduced in detail.

3.1 Observing Test Output in Different Layers

We begin by differentiating three layers, illustrated in Figure 3.1 of a user-interactive software application: the *Code Layer*, the *Behavioral Layer* and the *User Interaction Layer*.

3.1.1 Code Layer

At the lowest level, we have the source code. It is common to measure code coverage at the statement, branch or path level to determine coverage of the underlying business logic. It is very commonly used in experimentation to determine the quality of a new testing technique. The coverage information from multiple runs of the same test (suite) can be compared to determine their flakiness.

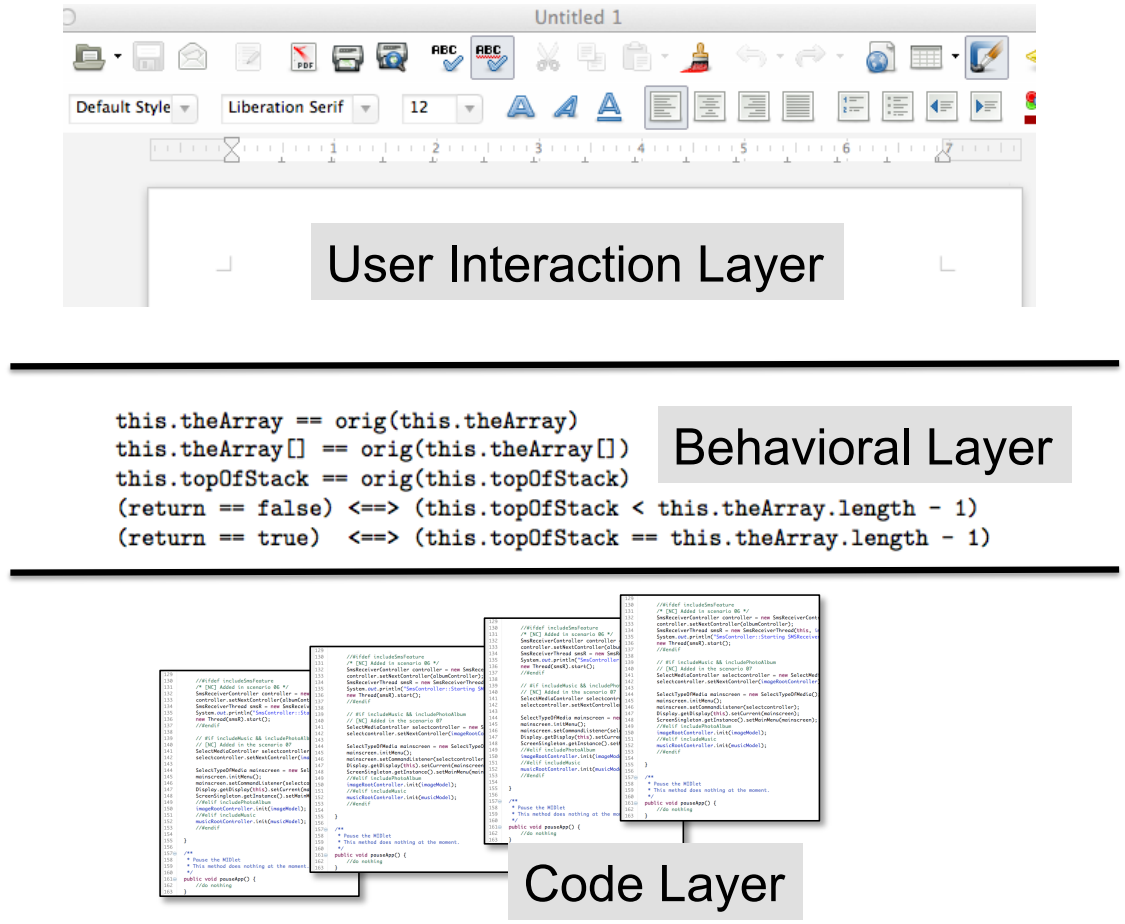


Figure 3.1: Layers of a User Interactive Application

3.1.2 Behavioral Layer

This layer presents behavioral data, such as runtime values of variables and function return values, and returning data to an external location or database, which may be obtained from the running program via instrumentation hooks. Such data can also be mined/analyzed to infer invariants. Invariant detection involves running a set of test cases and inferring from this, a set of properties that hold in all test cases. Invariants represent high-level functionality of the underlying code and should be consistent between runs. Test flakiness is measured at this layer by comparing

the invariants obtained from different runs.

3.1.3 User Interaction Layer

We use this to interface with the application and run tests. For example, in the Android system, one could write the code segment (shown in Figure 3.2) in a system test to programmatically discover and click on the "OK" button. During the test execution, the GUI state can be captured, and the flakiness at this layer can be computed.

```
UiObject okButton = new UiObject(
      new UiSelector().text("OK"));
okButton.click();
```

Figure 3.2: Code Segment from An Android Test Script

```
solo = new Solo(getInstrumentation(), getActivity());
solo.clickInList(1);
assertTrue(solo.searchText("Android")); // Assertion
```

Figure 3.3: An Example Test Oracle for An Android Application

We also can use this layer as test oracles to identify faults since this is the layer that the user sees. For instance, if a list widget fails to render or displays the wrong information, then this layer will reveal a fault. From an automation perspective, the properties of the interface widgets can be captured and compared to a correct

expected output, e.g., using the code shown in Figure 3.3 that (1) gets a handle to the current screen, (2) clicks on a list, and (3) checks whether the list contains the text "Android".

3.2 Dependent Variables: Metrics

Entropy was originally introduced in thermodynamics and has been used to measure the amount of disorder in a thermodynamic system. It was later adopted by information science and has some successful user scenarios [65–67], among which decision tree generation is one outstanding example [68] that utilizes the entropy metric to measure the *information gain*. The intuition behind this is that the entropy metric can be used to measure the amount of disorder of a data set, and the difference in entropy values reflects the additional or gained information that further ordered the data set.

We now describe the metrics to measure the variance of multiple runs of the same test. In addition to the entropy-based metric which measures test flakiness, we also introduce a novel metric to measure the *range of difference*. The metrics will be illustrated using the tests' code coverage information.

Assume we have the test cases with the coverage metrics as shown in Table 3.1. The coverage can be line or branch or some other unit (in our experiments we will use line coverage). The test suite (TS) includes 4 test cases (rows TC1-TC4), is executed 4 times on a subject application that has 6 lines of code (cols 1..6 within each run). A dot in the table means that the line is covered during a single

execution of the test case. For example, in the first run of test case 1 (TC1), lines 2 and 3 are covered whereas all other lines are not.

Table 3.1: Example Test Case/Suite Coverage Variance

#Cov	Run 1						Run 2						Run 3						Run 4						Const Groups
	1	2	3	4	5	6	1	2	3	4	5	6	1	2	3	4	5	6	1	2	3	4	5	6	
TC1		●	●					●	●					●	●					●	●				4
TC2	●	●		●			●	●		●	●		●	●		●	●		●	●		●	●		3/1
TC3		●			●			●			●		●	●		●	●		●	●		●	●		2/2
TC4		●	●	●				●	●	●				●		●				●	●	●		●	2/1/1
TS	●	●	●	●	●		●	●	●	●	●		●	●	●	●	●		●	●	●	●	●	●	3/1

Definition 2 (Consistently Covered Lines) *A line is* Consistently Covered *in two runs* iff *the line is covered in both runs or not covered in either of the runs.*

In our example, Line 1 is consistently covered in Run 1 and Run 2 of TC1 because Line 1 is not covered in either run. Line 2 is also consistently covered because it is covered in both runs.

Definition 3 (Consistent Coverage) *Two runs of a test case or test suite have* Consistent Coverage iff *all lines of the subject application are consistently covered in both runs.*

Both Run 1 and Run 2 of TC1 cover line set {#2, #3} and thus have consistent coverage.

Because the consistent coverage relationship is *reflexive*, *symmetric*, and *transitive* and is hence an *equivalence relation*, we can divide all runs of a test case/suite into equivalence groups based on the consistent coverage relationship, and measure **flakiness** of a test case/test suite as the *entropy* (or H) of the group based on the following formula [66]:

$$H(X) = -\sum_{i=1}^{n} p(x_i) log_e(p(x_i)) \tag{3.1}$$

where n is the number of groups and $p(x_i)$ is the probability that a certain run falls into the i^{th} group.

For example, the 4 runs of TC1 are divided into a single equivalence group of consistent coverage; thus we have $H(X)_{TC1} = -(4/4 * log_e(4/4)) = 0$. The 4 runs of TC4, however, are divided into 3 groups with sizes 2, 1 and 1, and the corresponding entropy is $H(X)_{TS} = -(2/4 * log_e(2/4) + 1/4 * log_e(1/4) + 1/4 * log_e(1/4)) = 1.04$.

To measure the impact of the variance, we further measure the *range of difference* of all runs of a test case/suite.

Definition 4 (Range of Difference) *The* Range *of all runs of a test case or test suite is the total number of lines that are not consistently covered in any two runs.*

For example, in the 4 runs of TC1, the same set of lines are covered in each run, thus the range of difference is 0, while the test suite has a range of 1 (at line #6). We show the ranges and entropies calculated for test cases and the test suite in Table 3.2. The average range and entropy of test cases are shown in the last row of the table. Note that the more unstable the groups are, the greater the entropy

value is. TC1 has perfect stability and thus has an entropy value 0. TC4 is the most unstable and thus has the greatest entropy value for test cases. The entropy of a test suite is generally smaller than the average entropy of all test cases. This is because some lines that are not covered by one test case may be covered by another and the individual differences are erased (we see this phenomenon in our study). If however, tests do not have this property, then the entropy of test suite will be higher than the average entropy of test cases.

Table 3.2: Entropy Metrics for Table 3.1

Metrics	Inconsistent Lines	Range	Groups	Entropy
TC1	{}	0	4	0
TC2	{#5}	1	3/1	0.56
TC3	{#1, #4}	2	2/2	0.69
TC4	{#3, #6}	2	2/1/1	1.04
TS	{#6}	1	3/1	0.56
Average		1.25		0.57

3.3 Summary

This chapter developed a formal way of quantifying test flakiness, i.e., in terms of entropy. The next chapter uses entropy on actual test runs to help identify factors that cause flakiness.

Chapter 4: Understanding Causes of Flakiness

4.1 Factors Impacting Test Execution

To better understand what causes tests to be flaky, this research categorizes the factors we believe have the greatest impact on test flakiness into four groups: test execution platform, application starting state/configuration, test harness factors and execution application versions.

4.1.1 Test Execution Platform

Many of the applications that we test can be run on different execution platforms which include different operating systems (e.g., Windows Mac OS, Linux) or on different versions of the same operating system (e.g. Ubuntu 12, Ubuntu 10). Different operating systems may render system interfaces differently and could have different load times, etc. In addition, applications often contain code that is operating system specific.

4.1.2 Application Starting State/Configuration

Many applications have preferences (or configuration files, registry entries) that impact how they start up. Research has shown that the configuration of an application impacts test execution [58] therefore we know that this starting point is important. Even if we always initialize with a default configuration at the start of testing, test cases may change the configurations for future tests. Therefore the program configuration files should be located and restored before each test is run.

4.1.3 Test Harness Factors

Test harnesses such as Selenium contain parameters such as *step delays* or *startup delays* to ensure that the application settles between test steps, however, this is often set to a default value and/or is set heuristically by the tester. Long delays may mean that the application pauses and runs additional code, but short delays may not allow completion of some functionality, particularly when system load or resources vary between executions. In early experiments we ran tests on a VM where we can change CPU and memory, and have found that reducing memory and/or CPU has a large impact on the ability to repeat test execution – primarily due to the need for tuning these parameters.

4.1.4 Execution Application Versions

If we are running web applications using different web browsers or Java programs using different version of Virtual machines (e.g. Java 7 versus 6 or OpenJDK

versus Oracle Java), we may run into differences due to support for different events and threading policies.

4.2 Empirical Study

We now evaluate the impact of factors that we have identified on test flakiness via the following research questions regarding the observed results at different layers:

- **RQ1:** To what extent do these factors impact code coverage?
- **RQ2:** To what extent do these factors impact invariant detection?
- **RQ3:** To what extent do these factors impact GUI state coverage?

Note that our questions include one for each of the layers. We start with the lowest code layer (code coverage) and end with the highest user interaction layer (GUI state).

4.2.1 Subjects of Study

We selected five non-trivial open source Java applications with GUI front ends from sourceforge.net. All of these have been used in prior studies on GUI testing. Table 4.1 shows the details of each application. For each we show the version, the lines of code, the number of windows and the number of events on the interface. `Rachota`[1] is a time tracking tool allowing one to keep track of multiple projects and create customized time management reports. `Buddi`[2] is a personal financial tool for

[1] http://rachota.sourceforge.net/en/index.html

[2] http://buddi.digitalcave.ca/

managing a budget. It is geared for those with little financial background. `JabRef`[3] is a bibliography reference manager. `JEdit`[4] is a text editor for programmers. Last, `DrJava`[5] is an integrated development environment (IDE) for Java programs.

Table 4.1: Programs used in Study

SUT Name	Version	LOC	# Windows	# Events
Rachota	2.3	8,803	10	149
Buddi	3.4.0.8	9,588	11	185
JabRef	2.10b2	52,032	49	680
JEdit	5.1.0	55,006	20	457
DrJava	20130901-r5756	92,813	25	305

4.2.2 Experiment Configurations

We selected our factors from each of the four categories based on those identified in Section 4.1.

1. **Platform:** We use the three operating systems that are described above (Ubuntu, Mac and Scientific Linux).

2. **Initial Starting State/Configuration:** We control the initial configuration of each application, input data files that are loaded or written to and when

[3]http://sourceforge.net/projects/jabref/

[4]http://sourceforge.net/p/jedit/

[5]http://www.drjava.org/

possible (on the Ubuntu and Mac OS stand-alone machines) we control the date and time of the machine running the tests (note that we could not control the time on our Red Hat cluster, but try to run tests within the same day when possible).

3. **Time Delay:** These are the delays that are used in the test harness to control the time that GUITAR waits to stabilize during replay of each step in the test case.

4. **Java Version:** We use three different Java versions in our experiments: Oracle JDK 6, Oracle JDK 7, and OpenJDK 6..

Our experiments vary each of the factors above. We do not vary all combinations of factors, but have designed a set of experiments that we believe is representative. The experiment configurations are shown in Table 4.2. Each configuration (row) represents a set of conditions.

Best means the best configuration (our gold standard for the experiments). For this we use the same configuration setup and use the same initial input files for the applications so that its starting state is the same. We also control the time (when possible) and fix the Java version to Oracle 6. To obtain the best configuration, we first tried to control as many factors as possible, and heuristically selected the best delay value for each different platform (where best shows the smallest variation based on a visual inspection of a sample of the test cases). We then fixed this configuration as our *best* configuration and created variants of these for study.

Table 4.2: Configurations of our Experiments

Runs	Config Fixed	Input Files	Date& Time	Delay	JDK
1. Best	Y	Y	Y	Best	Oracle 6
2. Unctrl	N	N	N (rand)	0ms	Oracle 6
3. No Init	N	N	N (actual)	best	Oracle 6
4. D-0ms	Y	Y	Y	0ms	Oracle 6
5. D-50ms	Y	Y	Y	50ms	Oracle 6
6. D-100ms	Y	Y	Y	100ms	Oracle 6
7. D-200ms	Y	Y	Y	200ms	Oracle 6
8. Opn-6	Y	Y	Y	Best	OpenJDK 6
9. Orc-7	Y	Y	Y	Best	Oracle 7

Unctrl means uncontrolled. This is expected to be our worst configuration. We do not control any of the factors mentioned. We just run our test cases with the default tool delay value (0ms), and do not reset the starting configuration files or provide a fixed input file. We use a random date (on the two platforms where we can control this).

Starting from our best configuration, we removed the initial configuration/starting state files. We call this **No Init** (see configuration #3). We then varied the delay values (shown as **D-0** through **D-200ms**, while keeping the configuration files

and inputs fixed. Our last configurations (**Opn-6** and **Orc-7**) use the best delays and control all other factors but use different versions of Java (Open JDK 6 and Oracle 7) instead of the default version (Oracle 6).

4.2.3 Experiment Procedure

Having selected the applications and setting up the configurations, our experiment procedure involves executing a number of test cases on these applications multiple times on various platforms and collecting information at the 3 layers discussed earlier. For each application we run 200 test cases randomly selected from all possible length 2 test cases generated by the GUITAR test case generator that are executable (and complete) on all three different platforms, Ubuntu 12.04, Red Hat Scientific Linux 6.4 and Mac OSX 10.8. The Ubuntu machine is a stand-alone server with an Intel Xenon 2.4GHZ CPU and 48 GB of memory. The MacOS machine is a laptop with a 2.5 GHZ Intel Core and 8GB of memory and the Red Hat Scientific Linux machine is an Opteron cluster server running at 2000MHz with 8GB of memory on each node. All test cases are short, which reflects what might be done for overnight regression or *smoke testing* [58]. We decided that using the shortest possible (reasonable) tests would keep us from unduly biasing this data. We view this as a baseline – if we can't control length 2 tests, then longer tests should be even harder. For each application and experiment configuration, we run the tests 10 times using the GUITAR replayer [56, 69].

We instrumented each application with the Cobertura code coverage tool [70]

to obtain the line coverage. We then parse the Cobertura report files to determine if we cover the same (or different) lines of code in each run. For invariant detection we use the Daikon Invariant Detector [64]. Due to the large number of invariants generated we only selected three classes for study, those with the highest code coverage. For interface oracles we use the states returned by GUITAR. For the oracle, we excluded common patterns that are known to be false positives (see [56]). More specifically, to avoid false positives and compare only meaningful properties of GUI state, we filter out properties, such as the ID of widgets given by the testing harness, minor differences in coordinates of widgets, etc. The remaining differences should be meaningful such as the text in a text field, or missing widgets. For the code coverage, we share the test cases and scripts between platforms to ensure consistency.

4.2.4 Addressing RQ1

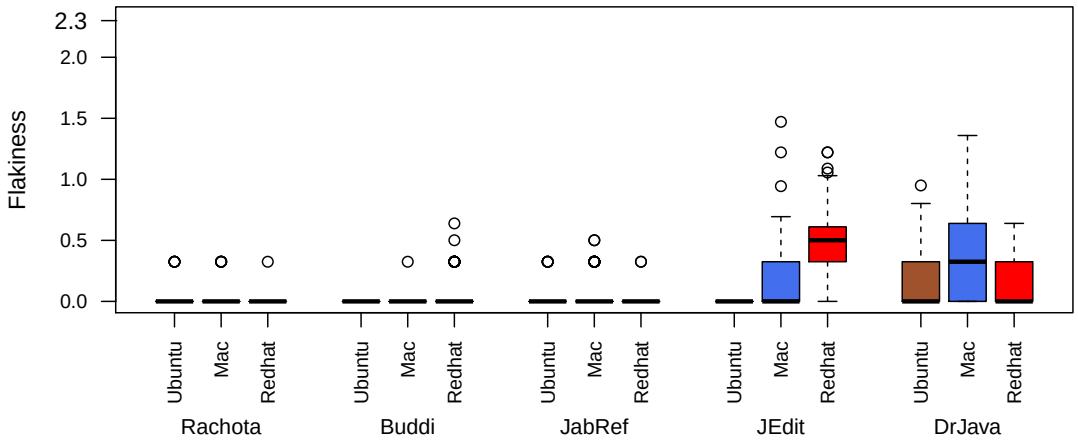
We begin by looking at the data for code coverage using the first and second configurations from Table 4.2. These configurations include the *best* with everything controlled and the potentially worst configuration (where we control nothing).

Table 4.3: Comparison Between Best & Uncontrolled Environments for Code Coverage and GUI False Positives

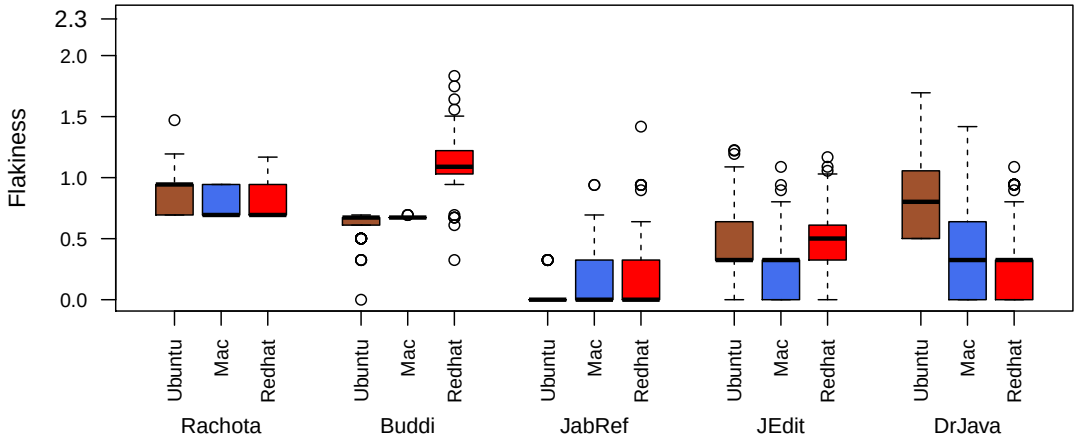
#Metrics		Rachota			Buddi			JabRef			JEdit			DrJava		
		Ubn	Mac	Rdh	Ubn	Mac	Rdh	Ubn	Mac	Rdh	Ubn	Mac	Rdh	Ubn	Mac	Rdh
TS	Best	0.67	0.69	0	0	0	1.23	0	0	0	0	0.33	0.33	0	0	0.64
	Unctrl	1.28	0.94	0.94	0	0.94	2.16	0	0	1.61	1.64	1.01	0.90	1.83	0.90	1.61
TC	Best	0.00	0.00	0.00	0	0.00	0.15	0.00	0.01	0.00	0	0.09	0.24	0.06	0.20	0.05
	Unctrl	0.42	0.40	0.41	0.32	0.34	0.57	0.01	0.06	0.10	0.22	0.10	0.23	0.41	0.18	0.13
Rg	Best	0.05	0.05	0.03	0	0.02	39.56	0.03	0.17	0.02	0	0.88	6.21	0.32	3.82	2.70
	Unctrl	83.85	80.22	83.09	4.98	5.00	90.83	2.72	2.72	181.62	72.63	5.03	1.99	184.22	13.19	13.37
FP	Best	4.44	1.44	2.11	0.06	0	1.56	0.50	0.06	0.50	0.11	3.17	1.50	0	5.67	0.17
	Unctrl	96.61	96.44	69.39	66.56	76.67	7.50	1.44	19.00	14.56	34.67	19.00	5.33	24.67	38.17	14.78

Table 4.3 shows the Test suite (TS) entropies for each application on each operating system. The rows labeled TC are the average entropy of the individual test cases within the test suite. In these tables we have 10 runs of test cases. Note that 0's without decimal places have true 0 entropy, while 0.00 indicates a very small entropy that is rounded to this value. The highest entropy occurs when all 10 runs differ – in this case we have an entropy of 2.3. When only a single run out of 10 differs the entropy is 0.33 and when half of the runs differ we have an entropy of 1.50. We see lower (close to zero), but not always zero, entropy when we control the factors, and higher entropy when we don't. We see differences between the applications and between platforms. We also show the range of coverage (in lines) which is the average variance across test cases. We can see that in the uncontrolled configuration we see as high as 184 lines on average differing and in the best we see closer to zero. However, we still have a few platforms/applications (such as Redhat running `Buddi`) where there is a large variance (90+ lines). This is because we were not able to control the time and date on this server and `Buddi` uses time in its code.

We show this data in an alternative view in Figure 4.1. The flakiness of all 200 test cases by application and operating system within the application. Figure 4.1(a) shows the best while (b) shows the uncontrolled factors. Flakiness is lower in the best environment (but not zero) across most of the applications and that it varies by platform. `Rachota` and `Buddi` have almost zero flakiness in their test cases while `JEdit` and `DrJava` have a non-zero flakiness even when we control its factors. Only the Ubuntu platform with the best configuration shows a zero flakiness.



(a) Best Controlled Environments



(b) Uncontrolled Environments

Figure 4.1: Flakiness Values of Line-Coverage Groups of 3 Platforms in Best & Uncontrolled Environments

Table 4.4: Flakiness Values of Test Suite Line Coverage for Initial State, Delays and Java Version

Flakiness	Rachota		Buddi		JabRef		JEdit		DrJava	
	Ubuntu	Redhat	Ubuntu	Redhat	Ubuntu	Redhat	Ubuntu	Redhat	Ubuntu	Redhat
No Init	1.42	0.69	0	0.94	0	0	0.33	0.67	0.80	1.09
D-0ms	0.69	0	0	0.80	0	0.64	0.50	0.64	0.64	1.28
D-50ms	0.94	0	0	1.47	0	0.33	0.33	0	0	0.64
D-100ms	0.69	0	0	1.19	0	0.33	0	1.06	1.17	1.09
D-200ms	0.67	0	0	0.50	0	0.33	0	1.03	0.94	1.36
Opn-6	0.94	0	0	1.47	0.33	0	0.50	1.36	0.95	0.90
Orc-7	1.36	0	0	1.23	0	0.33	0.80	1.61	0.64	0.94

Table 4.5: Average Flakiness Values of Test Case Line Coverage

Flakiness	Rachota		Buddi		JabRef		JEdit		DrJava	
	Ubuntu	Redhat	Ubuntu	Redhat	Ubuntu	Redhat	Ubuntu	Redhat	Ubuntu	Redhat
No Init	0.41	0.35	0	0.03	0.00	0.00	0.13	0.24	0.20	0.05
D-0ms	0.01	0.06	0	0.03	0.00	0.08	0.01	0.24	0.06	0.04
D-50ms	0.01	0.06	0	0.05	0.01	0.00	0.01	0.23	0.06	0.13
D-100ms	0.01	0.00	0	0.04	0.01	0.00	0.02	0.23	0.07	0.14
D-200ms	0.00	0.00	0	0.05	0.00	0.00	0	0.24	0.39	0.36
Opn-6	0.02	0.00	0	0.05	0.13	0.00	0.02	0.26	0.16	0.13
Orc-7	0.02	0.00	0	0.05	0.01	0.00	0.02	0.27	0.36	0.40

Table 4.6: Average Variance-Ranges of Line Coverage

Range	Rachota		Buddi		JabRef		JEdit		DrJava	
	Ubuntu	Redhat	Ubuntu	Redhat	Ubuntu	Redhat	Ubuntu	Redhat	Ubuntu	Redhat
No Init	85.19	63.41	0	84.59	0.02	4.06	10.06	1.92	14.06	4.89
D-0ms	0.10	0.72	0	94.18	5.69	168.46	0.19	7.63	0.93	1.49
D-50ms	0.12	0.71	0	150.65	2.61	6.46	0.13	1.77	0.32	12.70
D-100ms	0.09	0.03	0	146.83	0.05	5.89	0.14	1.92	0.97	14.12
D-200ms	0.05	0.03	0	114.62	0.03	5.18	0	2.00	82.77	14.63
Opn-6	0.25	0.08	0	43.08	0.80	3.56	0.22	56.75	1.55	13.94
Orc-7	0.38	0.03	0	13.68	0.06	11.29	7.36	61.01	97.45	15.56

We next turn to the configurations that do not control the initial environment (No Init), vary the delay time (D-$x$ms) and vary Java versions (Opn-6 and Orc-7). We show this data in Tables 4.4 through 4.6. Table 4.4 shows the test suite flakiness, while Table 4.5 shows the average test case flakiness. Table 4.6 shows the variance in line coverage. We see that the initial starting state and application configuration (No Init) has an impact on some of our applications, but not as large as we expected, when we control the other factors.

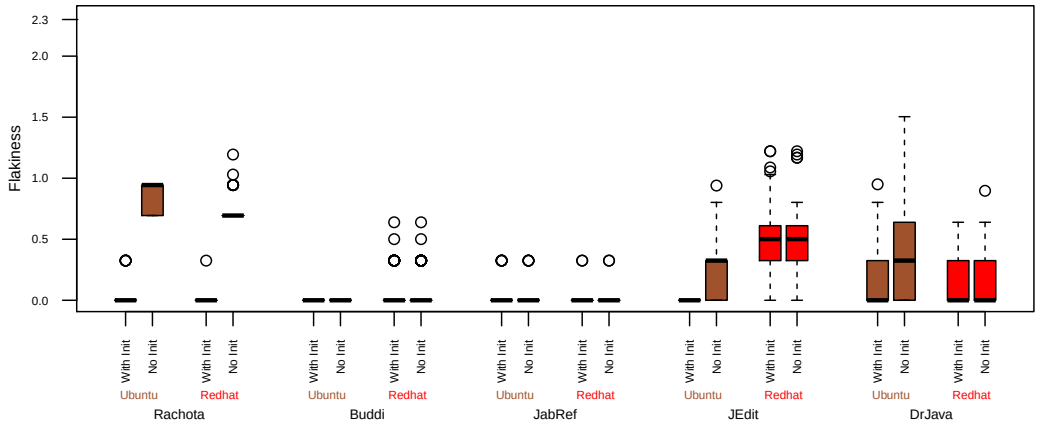


Figure 4.2: Flakiness Values of Test Cases of 2 Platforms with/without Initial State and Configuration Control

A boxplot of the flakiness values by application when we don't control the initial state and configurations is seen in Figure 4.2. We show a boxplot of the flakiness values by application for different delays in Figure 4.3. We see that the different delays also impact the flakiness and the delay value varies by application and platform. Finally, looking at the boxplots of the Java Versions in Figure 4.4, we

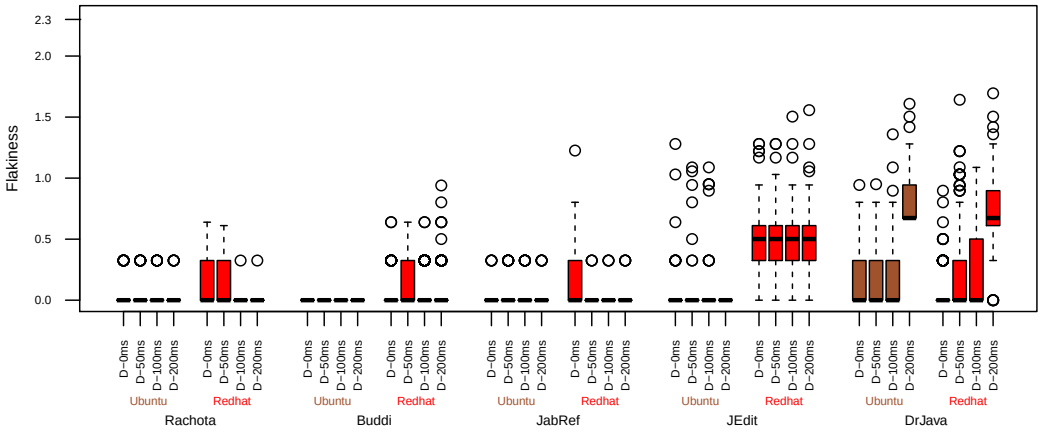


Figure 4.3: Flakiness Values of Test Cases with Different Delay Values

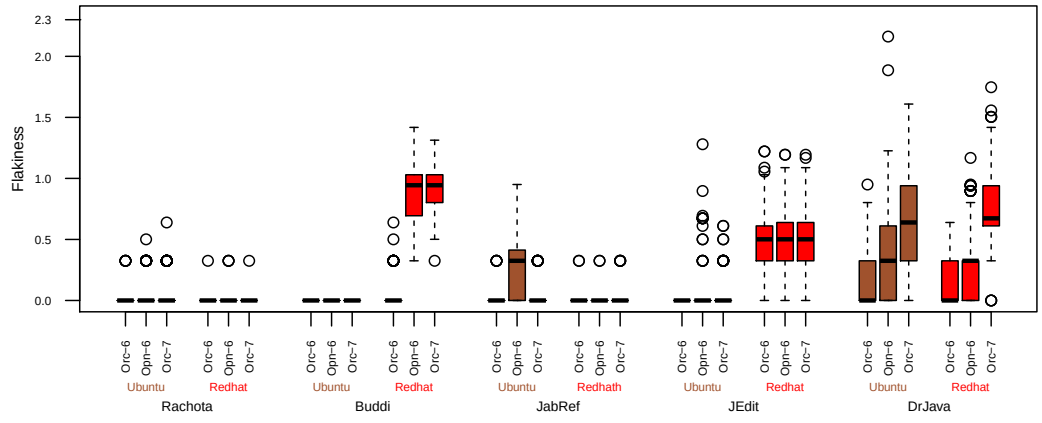
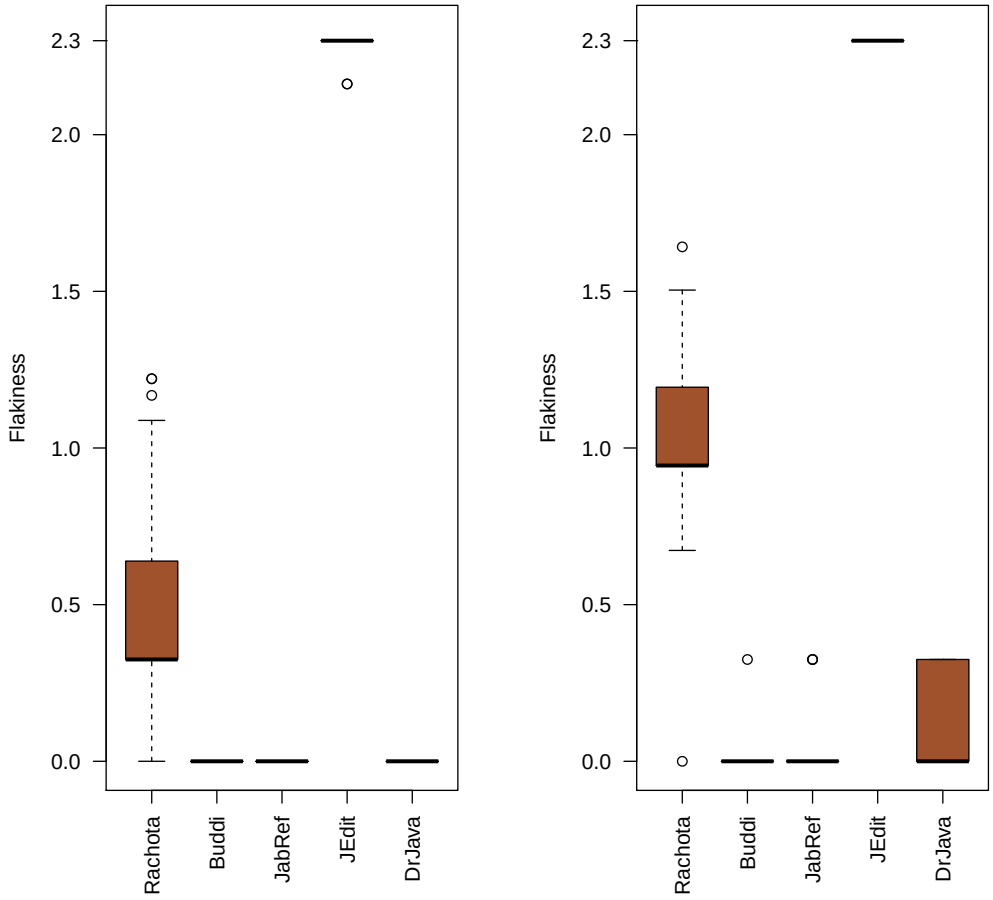


Figure 4.4: Flakiness Values of Test Cases with Different JDK Versions

see differences between Java versions, the largest being with `DrJava` using Oracle 7.



(a) Best Controlled Env

(b) Uncontrolled Env

Figure 4.5: Flakiness Values of Invariant Groups on Ubuntu

4.2.5 Addressing RQ2

To examine the results at the behavioral or invariant level we examine the invariants created by Daikon. Two runs of a test case have the same behavior if all the invariants that hold in these two runs are exactly the same. The average,

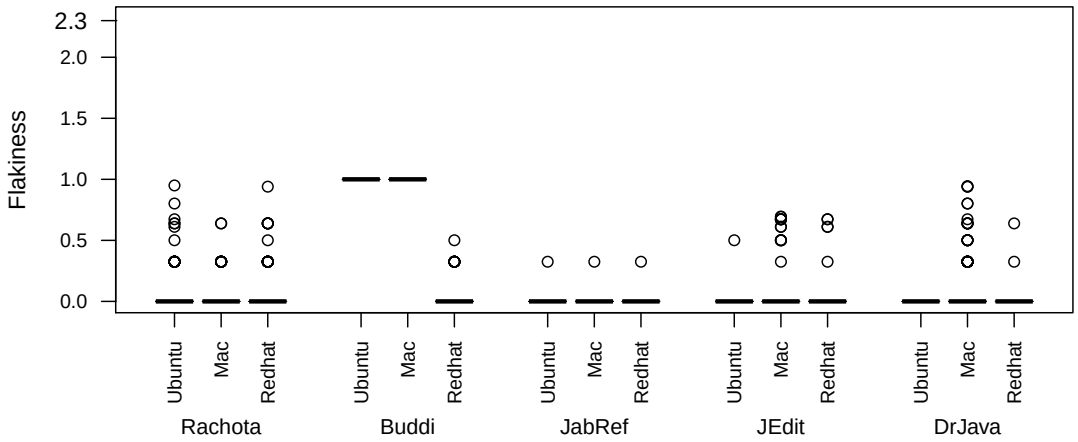
maximum and minimum flakiness values of the 200 test cases using the best and uncontrolled configurations are shown in Table 4.7. As we can see, the invariants seem to be more sensitive to application than to the factors that we are controlling. We can see this if we examine Figure 4.5. `Rachota` seems to have internal variation not related to these factors, while three of the other applications appear to have almost zero flakiness for both the best and uncontrolled runs. For `DrJava`, the factors impact the invariants. In general though the variation is lower than at the code level of interaction.

Table 4.7: Average/Max/Min Flakiness Values of Invariants across 200 Test Cases

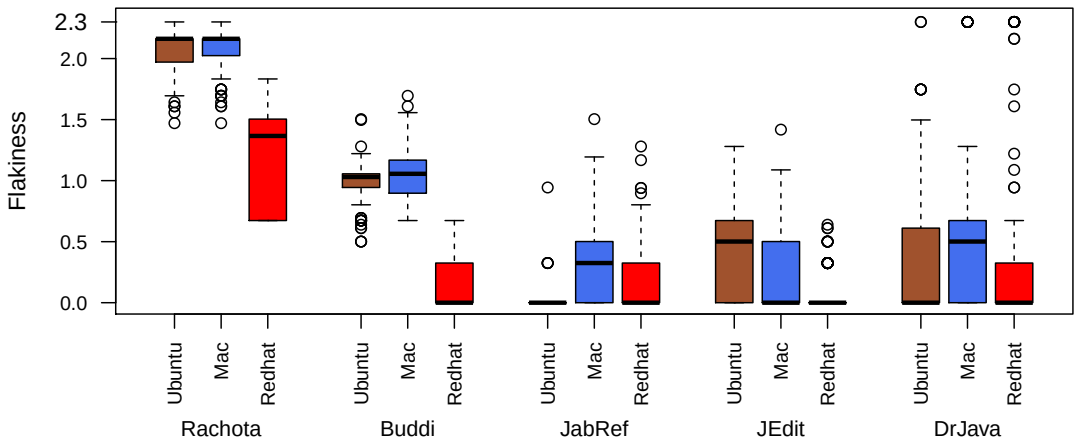
Config	Entropy	Buddi	Rachota	JEdit	JabRef	DrJava
Best	Avg	0	0.45	2.30	0	0
	Max	0	1.22	2.30	0	0
	Min	0	0	2.16	0	0
Unctrl	Avg	0.00	1.04	2.30	0.00	0.15
	Max	0.33	1.64	2.30	0.33	0.33
	Min	0	0	2.30	0	0

4.2.6 Addressing RQ3

For GUI layer, we capture the GUI state after each step of test execution, and after filtering out the spurious properties, we compare the GUI states from different



(a) Best Controlled Environment



(b) Uncontrolled Environment

Figure 4.6: Flakiness Values of GUI-State Groups of 3 Platforms in Best & Uncontrolled Environments

runs and obtain flakiness values. Figure 4.6 shows the distribution of flakiness values for GUI state for the 200 test cases. Figure 4.6(a) shows the flakiness value is very low for all 5 applications in the best controlled configuration, but that in 4.6(b) there is a higher median flakiness value when we leave our factors uncontrolled.

Since the GUI states are often used as test oracles for SUITs, we also measure the *false positives* of the test outputs. Our reasoning is that if one were to use the state as an oracle for fault detection, any change detected would indicate a fault. We use the state that is captured during the first (of 10) runs as the oracle, and then then calculate the (test level) false positives based on the following formula:

$$FP = \sum FalsePositives / (\sum \#testcases * (\#runs - 1)) * 100. \tag{4.1}$$

The results of false positives are shown in Table 4.3 in the last rows as **FP**. We see as high as a 96% chance (`Rachota` on Ubuntu) for obtaining a false positive. In general in the best configuration we see a very low false positive rate (no more than 6%), however it is only 0 in a few cases (such as `Buddi` on Mac). The high false positive rate is concerning for experiments that report on new techniques and finding faults. This data also concurs with other recent work on flakiness which uses fault detection as the metric (see Luo et al. [34]).

Finally we show the false positives for the other experimental configurations in Table 4.8. The results show that this level of information is sensitive to the initial state, delay values, and Java version.

Table 4.8: (Potential) False Positives detected by GUI State Oracle (%)

FP(%)	Rachota		Buddi		JabRef		JEdit		DrJava	
	Ubuntu	Redhat	Ubuntu	Redhat	Ubuntu	Redhat	Ubuntu	Redhat	Ubuntu	Redhat
No Init	48.17	38.78	0.22	2.94	0	1.00	2.22	0.28	10.22	11.33
D-0ms	2.22	3.06	0.06	4.22	1.83	12.28	2.50	4.61	0.22	3.44
D-50ms	2.44	5.94	0.50	1.56	27.33	1.06	0.72	1.17	0	1.61
D-100ms	1.22	9.33	0.22	1.57	1.00	0.61	0.22	0.06	0	3.72
D-200ms	4.44	1.28	0	2.56	0.50	0.72	0.11	0.06	65.22	2.39
Opn-6	3.39	2.61	1.78	22.22	0	0.56	0.39	0.44	0.83	3.44
Orc-7	2.83	3.83	26.61	23.89	0	1.83	17.67	1.22	0.17	3.11

4.2.6.1 Case Study: What Causes the Differences

Code Coverage: In our experiments, we found numerous instances of the same test case executing different code. Figure 4.7 shows an example of memory dependent code from the application `DrJava` that we cannot deterministically control with our test harness. In this code (lines 571-580 of StringOps.java, the *memSizeToString()* method) we see code that checks memory usage so that it can create a string object stating the memory size. It checks for whole block boundaries, e.g., 1B, 1KB, 1MB, via `whole == 1` and constructs the string content accordingly. Because the actual memory allocated to the executing JVM may vary from one run to the next, in our experiments one in ten executions covered this code because by chance the space was not equal to a block.

```
if (whole == 1) {
    sb.append(whole);
    sb.append(' ');
    sb.append(sizes[i]); ...}
```

Figure 4.7: An Example of Memory Dependent Code

The code segment shown in Figure 4.8 is an example of code that we can control by making sure the environment is mimicked for all test cases. In this code (`JEdit`, *MiscUtilities* class lines 1318-1357) the application checks file permissions. We found that the files did not always properly inherit the correct permissions when moved

by the test script and this caused 9 different lines being covered between executions, or when time and date were involved as has been described in [57]. Other examples of differing code coverage occurred when opening screens of windows have code that waits for them to settle (based on a time).

```
public static int parsePermissions(String s) {
   int permissions = 0;
   if (s.length() == 9) {
      if (s.charAt(0) == 'r')
          permissions += 0400;
      if (s.charAt(1) == 'w')
          permissions += 0200;
      ....
      else if (s.charAt(8) == 'T')
          permissions += 01000;
   }
   return permissions;
}
```

Figure 4.8: An Example of Environment Dependent Code

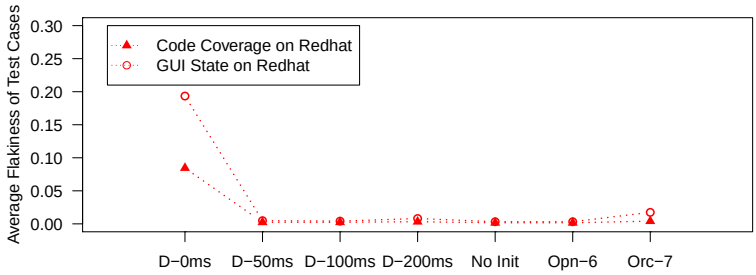
Invariants: In our experiments, we found differences in the invariants reported. For instance, in the application `Rachota`, we found that approximately in two of every ten runs, the application started faster than normal and generated the

two extra invariants shown in Figure 4.9 related to the startup window not found in the other eight runs. In each run we used exactly the same set of test cases. The "correct" set of invariants is dependent on the speed at which the window opens and again, may be an artifact of the system load or test harness factors.

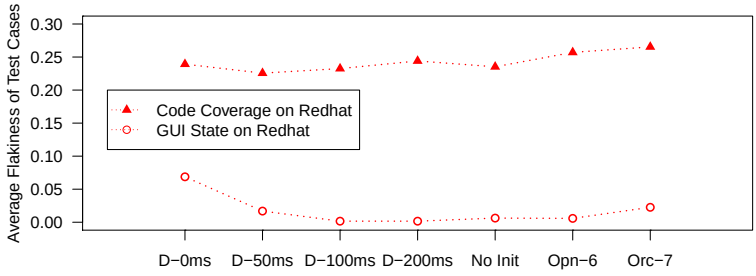
```
this.lbImage == orig(org.cesilko.rachota.gui
      .StartupWindow.startupWindow.lbImage)
this.loading == orig(org.cesilko.rachota.gui
      .StartupWindow.startupWindow.loading)
```

Figure 4.9: Example Differences in Invariants

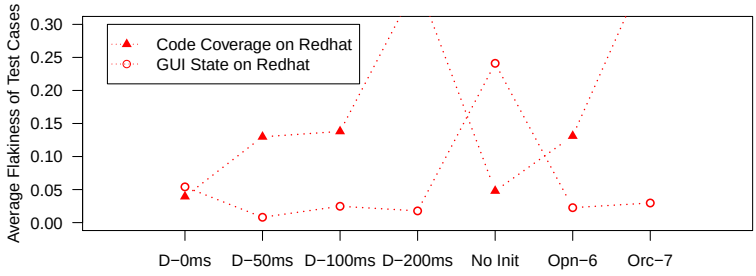
GUI State: During our experiments, we find variation in the properties of certain interface widgets between runs, that would appear as a fault to the test harness (and would be a real fault if this happened during manual execution), but that is most likely an artifact of automated test execution – a false positive. For instance in the application, `JEdit`, we found that the widget *org.gjt.sp.jedit.gui.statusbar.ErrorsWidgetFactory$ErrorHighlight* had empty text in one run, but was filled in during other runs. We believe this had to do with a delay between test steps that was too short when the system load increased, preventing the text from rendering to completion before the next test step occurred. A test case that checks the text in this widget may non-deterministically fail during one run and succeed in another.



(a) JabRef on Redhat



(b) JEdit on Redhat



(c) DrJava on Redhat

Figure 4.10: Average Flakiness: Code Coverage vs GUI State

4.2.6.2 Discussion: Correlation Between Code Coverage and GUI State

To study a possible correlation between the stableness of code coverage and GUI state layers, we plot curves of average flakiness in Figures 4.10(a) - 4.10(c), for 3 applications on the RedHat platform. We see that the flakiness in code coverage is generally greater than the GUI state. But we don't see a correlation between the two. Figure 4.10(b) shows a big difference in code coverage and GUI state flakiness; sometimes code coverage is unstable when GUI states are stable. Figure 4.10(c) shows this trend.

4.3 Discussion and Guidelines

We have seen some interesting results during these experiments. First, in almost none of our layers were we able to completely control the information obtained. We saw instances of an application with zero entropy for one or two of the testing configurations, but in general most of our results had a positive entropy meaning at least one test case varied. Some of the greatest variance appears to be related to the delay values and issues with timing which are system and load dependent. Since this might also be an artifact of the tool harness that we used (GUITAR), we wanted to understand the potential threat to validity further, therefore we ran an additional set of experiments using the Selenium test case tool which automates test execution on web applications.

We selected a web application that has existing test cases called `schoolmate` version: 1.5.4. It was used as one of the subjects by Zou et al. in [71]. `SchoolMate` is a PHP/MySQL solution for elementary, middle and high schools. We randomly selected 20 of the test cases and modified the delay values of the test cases to be 0ms, 100ms and 500ms. We ran each 10 times as we did in our other experiments. Six of the twenty test cases failed at least once (and 5 failed in all 10 runs) when the delay value was 0ms or 100ms. This shows us that the delay value is relevant in other tools as well.

One might expect monotonic behavior with respect to the delay, but we did not observe this. Since tools such as GUITAR cannot use human input to advise them about which properties to wait for, they use heuristics to detect an application steady state. The delay value says how long to wait before checking for a steady state. In some applications, there are system events that check periodically for a status, such as timing or reporting events. Since these run at intervals (and the intervals may vary), they create a complex interaction with the delay value, resulting in unpredictable behavior.

We also found some interesting application specific issues such as code which is dependent on the size of memory the application is using, the time of day that the application is run or the time it takes to refresh a screen. For instance, one application had a 30 day update mechanism and we just happened to run one of our early experiments on the 30 day update (and covered new/additional code). Had we been running a real experiment, we might have incorrectly reported that our technique was superior. With respect to operating system differences, we saw three

primary causes. Some differences are due to permissions or network connections. For instance, `Rachota` will send requests to a server using the network and these calls were blocked on the Mac experiments. We also saw differences with system load. Certain code is triggered if the application takes longer to perform a particular task or if the resolution is different. This is not necessarily due to differences in the operating systems, but is machine specific. Last, we found code such as in `JabRef` that only runs under specific operating systems enclosed within `if`-blocks.

We did find that we could control a lot of the application starting state and configurations by judicious sharing of the starting configuration files, and that if we heuristically find a good delay value for a specific machine it is relatively stable. The invariant layer was our most stable layer, which indicates that the overall system behavior may not be as sensitive to our experimental factors. Despite our partial success, we believe there is still a need to repeat any tests more than once.

Our overall guidelines are:

1. **Exact configuration and platform states must be reported/shared**. When providing data for experimentation or internal company testing results, the exact configuration information, including the operating system, Java version, harness configurations/parameters and the application starting state needs to be shared.

2. **Run Tests Multiple Times.** For some of the differences observed we do not see an easy solution and expect some variance in our results. Therefore studies should include multiple runs of test cases and report averages and variances of

their metrics, as well as sensitivity of their subjects to certain environmental conditions (e.g., resources, date/time).

3. **Use Application Domain Information** to help reduce some variability. For instance, if the application uses time, then clearly this is an environmental variable that needs to be set. But we found others such as memory variances, file permission variances, and simple timing delays within the application that would vary. Knowing what some of these are may allow you to remove that variable code from the evaluation.

4.4 Summary

This chapter studied the impact of factors, such as test execution platform, SUT initial state, and timing, on flakiness of test cases. The results showed that by controlling these factors properly, test flakiness can be largely reduced. But still, we cannot totally eliminate flakiness with the best controlled environment in our study - flaky failures may still be reported. In the next chapter, we present a flake filter that filters out flaky failures and retains fault detection ability of tests in the meanwhile.

Chapter 5: Minimizing Effects of Flakiness

Flaky tests are often reported as failed and thus require additional resources for re-execution. This research alleviates this problem by developing a new *flake filter* that automatically weeds out flaky failures. We thus study the effectiveness of the flake filter via the following research questions:

- **RQ4:** What is the impact of flakiness on test results?
- **RQ5:** How effective is our flake filter?
- **RQ6:** What is the cumulative effect of applying the flake filter over multiple successive versions of an application?

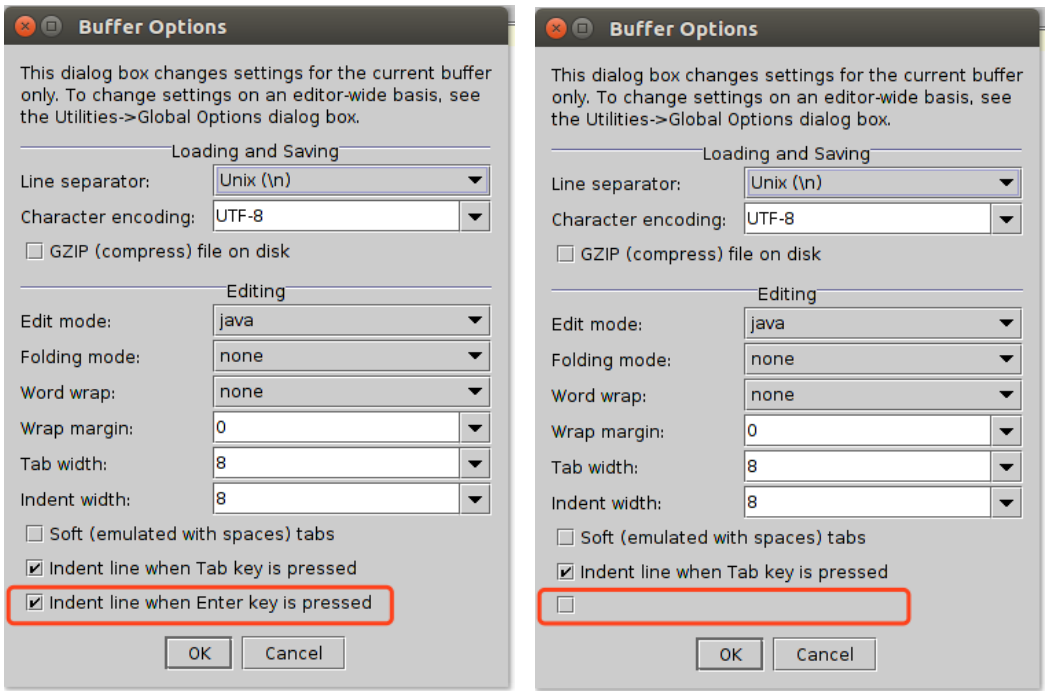
5.1 Developing Our Flake Filter

We now describe the design and development of our flake filter using a running example. Consider the test case shown in Figure 5.1. This is an actual test case from our empirical study (Bug 1324 of the jEdit application from Section 5.2). The test case contains two parts. The first part includes 2 actions that will be executed on the SUT: in the main window (titled "jEdit") of the SUT, the test clicks menu "Utilities" and then menu item "Buffer Options...". As a result, a dialog entitled

```
// Click on the menu item to invoke the "Buffer Options"
   dialog.
Window("jEdit").JMenu("Utilities").Click
Window("jEdit").JMenuItem("Buffer Options...").Click


// Assertions on the SUT state.
Dialog("Buffer Options").
      CheckProperty "isRootWindow", "false" // asr1
Dialog("Buffer Options").
      CheckProperty "width", "358"  // asr2
Dialog("Buffer Options").
      CheckProperty "height", "496" // asr3
Dialog("Buffer Options").Checkbox(#4).
      CheckProperty "text", "Indent..." // asr4
Dialog("Buffer Options").Checkbox(#4).
      CheckProperty "isSelected", "true" // asr5
Dialog("Buffer Options").Checkbox(#4).
      CheckProperty "y", "22" // asr6
```

Figure 5.1: Example Test Case



(a) V_1 without the Bug (b) V_2 with the Bug

Figure 5.2: Bug 1324 of the JEdit Application

"Buffer Options" opens as shown in Figure 5.2(a). The second part of the test case includes 6 assertions that check the correctness of the SUT after executing the actions. In the example, 3 assertions check properties "isRootWindow", "width" and "height" of the dialog, and 3 assertions on properties of the highlighted checkbox, "text", "isSelected" and "y" (y coordinate).

The test was originally developed on the current version, V_1, of the SUT; we executed it and it passed on V_1 at least once. When the SUT was changed to a new version, V_2, a bug is introduced in the "Buffer Option" dialog. As shown in Figure 5.2(b), the text label for the checkbox mistakenly disappeared. In addition, the checkbox was incorrectly not selected. Thus we expect assertions *asr4* and *asr5* to fail on V_2, thereby revealing the bug.

Table 5.1: Test Results on New Version

Assertion	Expected	Actual	Status	As Expected	Flakiness
asr1	false	false	*Passed*	*Yes*	0
asr2	358	348	**Failed**	**No**	**1.03**
asr3	496	498	**Failed**	**No**	**1.28**
asr4	"Indent..."	""	*Failed*	*Yes*	0
asr5	true	false	*Failed*	*Yes*	**0.33**
asr6	22	17	**Failed**	**No**	**0.61**

The results of executing the test case on V_2 are shown in Table 5.1. For each assertion, the first two columns show the "Expected" and "Actual" values of the object properties returned to the assertions. In the "Status" Column, we mark the assertion as *Passed* if the two values match, and **Failed** otherwise. Among the 5 failed assertions, not all were expected/wanted. Assertions *asr4* and *asr5* are as expected, whereas other failures were not.

We have defined *flaky tests (suites)* in Chapter 1. Now we formally define flaky assertions and objects.

Definition 5 (Flaky assertions) *An test assertion is flaky if it does not consistently pass or fail across multiple runs on the same SUT with the same configuration.*

Test assertions often assert some property values on a certain program objects. For example, assertion *asr4* is performed on a *checkbox object* with id number *#4*

inside a dialog titled "Buffer Options". Thus we define *Flaky objects* as below.

Definition 6 (Flaky objects) *An program object is flaky if at least one of its properties/attributes returns unexpectedly different results across multiple runs of the same test on the same SUT with the same configuration.*

Note that according to our definitions, when an assertion is flaky, the object asserted on is also flaky. And a test is flaky as long as one or more of its assertions are flaky.

5.1.1 Quantifying Flakiness via a Score

Our flake filter discards the outcomes of any assertion that causes a certain level of flakiness in each test case. We quantify flakiness of an assertion, a, by measuring entropy of the actual observed values of each test assertion during multiple runs using the following formula:

$$\mathcal{F}(a) = H(X) = -\sum_{i=1}^{n} p(X_i) log_e(p(X_i)) \tag{5.1}$$

where X_i stands for an identical value observed on the asserted object property and $p(X_i)$ stands for the probability of the object having property value X_i.

In this example, we run the test case on V_1 10 times and keep record of the actual values of the properties asserted on. For example, we observed 3 identical values of the *width* property used in *asr2*: 358 (5 times), 348 (3 times) and 338 (2 times). Thus $\mathcal{F}(asr2)$ can be calculated as:

$$-(\tfrac{5}{10} * \log_e(\tfrac{5}{10}) + \tfrac{3}{10} * \log_e(\tfrac{3}{10}) + \tfrac{2}{10} * \log_e(\tfrac{2}{10})) = 1.03$$

As another example, we observed 2 identical values of the *selected* property used in *asr4*: true (9 times) and false (1 time). Thus $\mathcal{F}(asr4)$ can be calculated as:

$$-(\tfrac{9}{10} * \log_e(\tfrac{9}{10}) + \tfrac{1}{10} * \log_e(\tfrac{1}{10})) = 0.33$$

From the examples, we can that the more unstable the observed value of an object property is, the greater its flakiness score will be. And as extreme cases, the smallest score, 0, will be obtained *iff* outputs in all runs are identical. On the other end, if all outputs in 10 runs are different, the greatest score, $-\frac{1}{10}\log_e(\frac{1}{10})*10 = 2.30$ will be obtained.

By applying our flakiness formula to all assertions, we can get their flakiness scores in the 10 runs, and the results are shown in the last column of Table 5.1.

We measure the flakiness of a test case as the maximum flakiness of all its assertions:

$$\mathcal{F}(test) = MAX_{asr_i \in test}\ \mathcal{F}(asr_i) \tag{5.2}$$

Thus the flakiness score of the test case is a positive number, 1.28, showing that the test is flaky. To obtain a non-flaky test, we can apply our filter to remove all flaky assertions, results in the test case shown in Figure 5.3.

5.1.2 Tuning the Threshold of Flakiness Score

In the resulting test case, we remove all flaky tests, thus making reported failures more reliable. But the problem is one of the bug-revealing assertion, *asr5*, is also filtered out. Thus we developed a technique to better balance between removing flaky assertions and retaining bug-revealing ones by tuning the threshold of flakiness

```
// Click on the menu item to invoke the "Buffer Options"
   dialog.
Window("jEdit").JMenu("Utilities").Click
Window("jEdit").JMenuItem("Buffer Options...").Click


// Assertions on the SUT state.
Dialog("Buffer Options").
      CheckProperty "isRootWindow", "false" // asr1
Dialog("Buffer Options").Checkbox(#4).
      CheckProperty "text", "Indent..." // asr4
```

Figure 5.3: Example Test Case After Filtering Out Flaky Assertions

score. Giving a flakiness threshold, τ, our flake filter will eliminate all assertions with flakiness scores greater than τ.

The results of applying different thresholds are shown in Figure 5.4. The dash line with hollow nodes show the percentage of flaky assertions eliminated - as the threshold increases, some not-so-flaky assertions will be retained, causing the line to decrease. The concrete line with filled nodes show the percentage of bug-revealing assertions retained - as the threshold increases, some flaky bug-revealing assertions will be retained, causing the line to increase. By tuning the threshold, we can achieve best performance of our flake filter based on our needs. For example, when we pick a threshold between 0.4 and 0.6, our filter can eliminate 75% of flaky assertions, and retain all (100%) bug-revealing assertions. The intersection of the two lines is

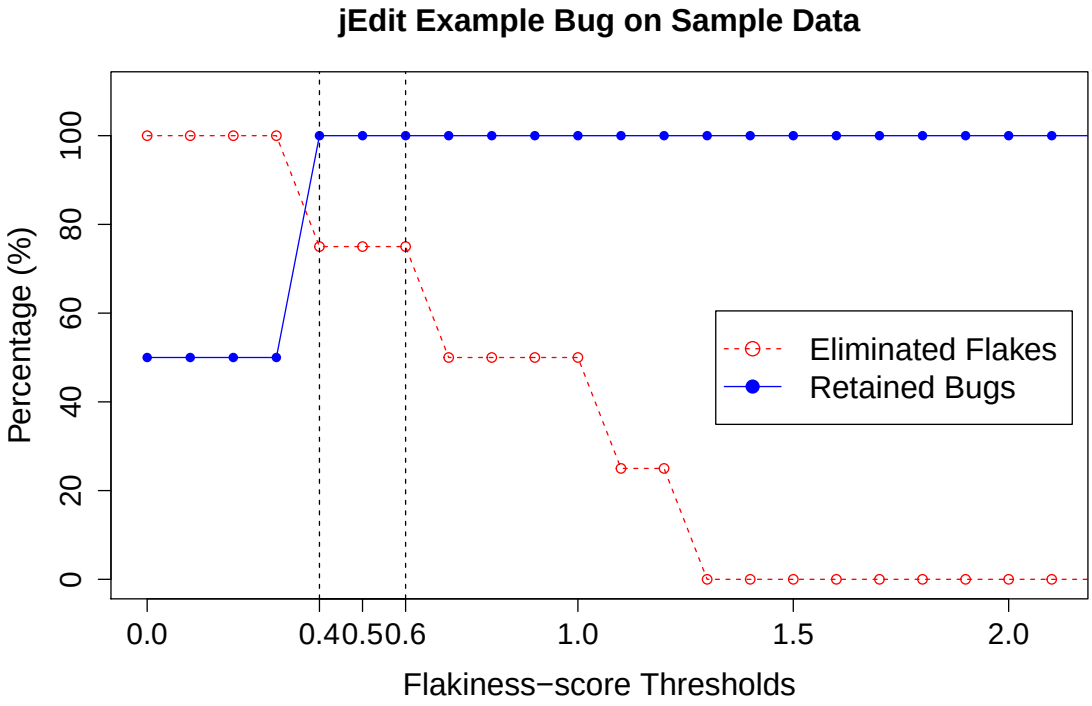


Figure 5.4: Performance of Flake Filter on jEdit Example Bug

roughly at coordinate (0.37, 0.83), indicating the balance point in our two goals: if we had more data, our filter is expected to eliminate 83% of flaky assertions and in the meanwhile retain 83% of bug revealing assertions when the threshold is set to 0.37. The real data for this bug is shown at the top of Figure 5.9 in Section 5.2.

5.1.3 Accumulating Flakiness Information Across Versions

In this part, we are going to test a new version, V_3, of the SUT, and want to apply our flake filter to eliminate failures on V_3 that were flaky on V_2.

Similarly, to apply our flake filter, we need to run the test case on V_2 multiple times to measure the flakiness of assertions in the test. The results are shown in Table 5.2. As shown in Column "Flaky on V_2", 2 assertions (*asr2* and *asr3*) were

Table 5.2: Test Results on V_3

Assertion	Flaky on V_2	Failed on V_3	Filtered Single-V	Filtered Accu
asr1	No	No	-	-
asr2	Yes	Yes	*Yes*	*Yes*
asr3	Yes	Yes	*Yes*	*Yes*
asr4	No	Yes	*No*	*No*
asr5	No	No	-	-
asr6	No	Yes	**No**	**Yes**

observed flaky on V_2. Column "Failed on V_3" shows 4 failed assertions. Besides of the two flaky failures, a true failure on assertion *asr4* reported the unfixed bug; but assertion *asr6* was a mis-reported failure that our flake filter failed to eliminate because it had not been observed flaky on V_2. Fortunately, the missing assertion, *asr6*, has been observed flaky on the history version, V_1. Thus we extend our flaky filter to accumulate flakiness information across versions. As shown in the last two columns of Table 5.2, our flake filter with flakiness information from a single version (V_1) filtered 2 flaky failures (*asr2* and *asr3*); and by accumulating flakiness information across multiple versions (V_1 and V_2), our flake filter eliminated 1 addition flaky failure (*asr6*), improving the performance of the single-version flake filter by 50% in this example.

5.2 Empirical Study

The goal of this study is to study the extent of flakiness in fielded software systems and their test suites, evaluate the usefulness of our flakes filter, i.e., its ability to reduce flaky failures, and explore the unintended consequences of missing real faults due to the flakes filter.

5.2.1 Metrics for Research Questions

Now we defined metrics to evaluate our research questions.

RQ4 addresses the impact of flakiness on test results. We will execute test cases, containing test steps and assertions on program objects, on a set of subject applications multiple times. We will record the outcome of each assertion and test case. Our metrics to answer this question will be percentage of flaky tests, flaky assertions and flaky objects. These metrics will inform us about the extent of flakiness that exists in our subject applications and their test suites.

Additionally, we will select, from a given universe of tests for a subject application, a subset that is capable of detecting a particular real fault in the application. We will then evaluate the flakiness of all assertions—if there are multiple—in that test case, including the one that caused the test case to fail, and eventually lead to the detection of the fault. This is important because the flakiness of these assertions may cause the test case to incorrectly pass, thereby causing the actual fault to be missed; or if the flaky assertions fail, the test case would fail for an incorrect reason; both are undesirable as they lead to wasted resources, e.g., manual triaging

of failure results. We will repeat this analysis for multiple faults, test cases, and subject applications. Hence, we will mine real subject applications for actual faults that have been reported for their test cases.

RQ5 addresses effectiveness of our flake filter. We will measure effectiveness in two ways: (1) the ability of the flake filter to reduce flaky failures and (2) the unintended consequence of missing real bugs due to our flake filter. Because our flake filter is enabled via a tunable *flakiness threshold*, i.e., we filter all assertions/tests whose flake score is greater than the threshold, the flake filter's effectiveness using our two aforementioned criteria is closely tied to the threshold: set too high (an underperforming filter) and we risk too large a number of flakes; set too low (an aggressive filter) and we risk missing real faults.

For ease of understanding, let's assume that, for each test case t_i, we have computed a flakiness score $\mathcal{F}(t_i)$. Further, assume the availability of a function $\Phi(t_i, \mathcal{F}(t_i), \tau)$, where $\tau \geq 0$ is the threshold, and returns NULL if $\mathcal{F}(t_i) > \tau$, else returns t_i. The way we have computed our flakiness score, $\Phi(t_i, \mathcal{F}(t_i), 0)$ will always return NULL if t_i has exhibited any amount of flakiness. Hence, our first metric to compute the effectiveness of our flake filter for a test suite $T = \{t_1, t_2, \ldots, t_n\}$, and threshold τ is $\mathcal{R}$:

$$\mathcal{R}_T(\tau) = \frac{|Map(\Phi(\#1, \mathcal{F}(\#1), \tau)\&, T)|}{|Map(\Phi(\#1, \mathcal{F}(\#1), 0)\&, T)|} \times 100, \tag{5.3}$$

where *Map(f&,L)* applies the function $f()\&$ to each element of a list L, substituting the arguments in $f()\&$ (denoted by #1) with the element, and returns a list of the outcomes of $f()\&$. Intuitively, the above formula gives us the percentage of flaky

tests that our flake filter can remove. Similar computations can be done for flaky assertions and objects, e.g., to compute the effectiveness of our flake filter for a set of assertions $A = \{a_1, a_2, \ldots, a_m\}$ for a test suite, we adapt the formula to:

$$\mathcal{R}_A(\tau) = \frac{|Map(\Phi(\#1, \mathcal{F}(\#1), \tau)\&, A)|}{|Map(\Phi(\#1, \mathcal{F}(\#1), 0)\&, A)|} \times 100. \tag{5.4}$$

Our second metric has to do with our filter inadvertently filtering out test cases and assertions that detect actual faults. Assume that we have for each test case t_i, a function $\beta(t_i)$ that returns 1 if t_i detects a real defect (not flaky) in the software under test, else returns 0. It also returns 0 on a NULL input parameter value. We compute our second metric $\mathcal{B}$ for test suite $T = \{t_1, t_2, \ldots, t_n\}$:

$$\mathcal{B}_T(\tau) = (1 - \frac{\sum Map(\beta(\Phi(\#1, \mathcal{F}(\#1), \tau))\&, T)}{\sum Map(\beta(\#1)\&, T)}) \times 100, \tag{5.5}$$

where $\sum Map(\beta(\#1)\&, T)$ gives us the *total* number of bug revealing test cases from amongst $\{t_1, t_2, \ldots, t_n\}$. $\sum Map(\beta(\Phi(\#1, \mathcal{F}(\#1), \tau))\&, T)$ gives us the total number of bug revealing test cases that were filtered out by the flake filter. Again the formula for $\mathcal{B}$ can be adapted for assertions instead of test cases by simply using the set of assertions A instead of T.

Because we want real regression failures as well as flaky failures to address this research question, we will select software subjects with at least two versions: V_{X-1} and V_X. We will select V_X as a version in which a regression bug was detected by at least one test case; and V_{X-1} that did not have that particular bug.

RQ6 addresses the cumulative effect of applying the flake filter over multiple successive versions of an application. Once a flake filter has been obtained for a

software version and test suite, one would expect that the same assertions would exhibit flaky behavior on subsequent versions of the software for the suite unless fixed. If one were to evolve the flake filter, adding newly identified flaky assertions/tests over time, the cumulative effect of the evolved flake filter should yield a smaller set of flaky tests than would have been obtained from a single version. To address this question, we will run tests on 10 consecutive versions of each subject application, obtain the set of flaky tests and assertions, constantly evolving the flake filter. At each step, we will show the cumulative effect of the flake filter on the resulting test suite.

5.2.2 Subjects of Study, Test Cases, & Bugs

We have used 3 Java applications in our experiments: jEdit, Jmol,[1] and JabRef. jEdit and JabRef have been used in our first empirical study, and Jmol is a molecular viewer for three-dimensional chemical structures. These applications have been used in our past studies, come with a set of test cases, and have real reported faults [36].

Table 5.3 shows the number of test cases for each of our subjects. Columns "Objects" and "Assertions" shows the number of objects and assertions observed during test execution.

The test cases for these applications detected a total of 16 regression bugs; we found these bugs from reports submitted at these applications' bug reporting sites

[1] http://sourceforge.net/p/jmol/

Table 5.3: Test Cases for Subject Applications

SUT	Version	Tests	Objects	Assertions
jEdit	4.1Pre4	100	908	33215
	4.3.3	73	1126	37514
	4.4.2	42	1200	25756
Jmol	11.6.27	16	498	10249
	12.2.34	35	1084	27892
	13.1.3	19	471	12707
JabRef	1.1	41	247	6277
	1.5	100	472	23849
	1.7.1	100	623	21327
Total	All	526	6629	198786

Table 5.4: Bugs in jEdit, Jmol and JabRef

AUT	Bug Id	Bug Description	V_{X-1}	V_X	V_{X+1}
jEdit	1324	CheckboxMissing	4.1-pre4	4.1-pre5	4.1-pre6
	3538	BeanShellError	4.3.3	4.4-pre1	4.4.1
	3645	DropdownlistUnenabled	4.4.2	4.5.1	5.0-pre1
	3899	DropdownlistEmpty	4.3.3	4.4-pre1	5.0-pre1
Jmol	T1	LogoInNewWindow	11.6.27	11.7.1	12.0.38
	T2	NewWindow	13.1.3	13.1.4	13.1.14
	T3	LogoInAboutWindow	12.2.34	13.0.1	14.2.11
	T4	MainWindowTitle	12.2.34	13.0.1	NA
JabRef	65	HelpContent	1.1	1.2	1.3.1
	160	SearchColumn	1.5	1.6-beta	1.7-beta2
	1130	CloseDatabase	1.7.1	1.8-beta	2.0-beta
	1132	SaveDatabase	1.7.1	1.8-beta	2.0-beta
	1133	Search	1.7.1	1.8-beta	2.0-beta
	1134	Export	1.7.1	1.8-beta	2.0-beta
	1135	EditEntry	1.7.1	1.8-beta	2.0-beta
	1136	SearchPanel	1.7.1	1.8-beta	2.0-beta

(mostly SourceForge[2]). Table 5.4 shows the bugs and the versions in which they first appeared. For each bug, we use a concise, descriptive keyword to describe each bug. We denote the version in which the bug first appeared as V_X, the past version in which the broken feature still worked as V_{X-1} and the future version in which the bug was first fixed as V_{X+1}.

In Tables 5.5 through 5.7, for each bug, we also show a sample test case that automatically revealed the bug as well as the cause of this revelation, i.e., the properties that mismatched, and hence triggered the test oracle to report a failure. The columns "Steps" and "TC length" show the steps and length of the test cases. The "Oracle" column shows the mismatch.

[2]http://sourceforge.net/

Table 5.5: jEdit Bugs

Bug Id	Steps	TC Length	Oracle
1324	1. Click on the tool bar icon “Buffer options”	1	A checkbox and a text label “Indent line when Enter key is pressed” are shown in V_{X-1} but missed in V_X.
3538	1. Click on menu “View”; 2. Select the menu item “Global Scope” under “Buffer Sets”.	2	A dialog “BeanShell Error” is shown in V_X.
3645	1. Click on tool bar icon ”Global options”; 2. Click on menu tree item ”Status Bar”; 3. Click on tab “Widgets”; 4. Click on button ”+”.	4	The “Enabled” property of the JComboBox “Choose a Widget” is True in V_{X-1} but false in V_X.
3899	1. Click on tool bar icon ”Global options”; 2. Click on menu tree item ”Status Bar”; 3. Click on tab “Widgets”; 4. Click on button ”+”.	4	The JComboBox “Choose a Widget” is empty in V_X, whereas an element, a BasicComboBoxRenderer titled “lineSep”, is shown in V_{X-1}.

Table 5.6: Jmol Bugs

Bug Id	Steps	TC Length	Oracle
T1	1. Click on menu "Help"; 2. Click on menu item "What's new".	2	The logo of Jmol is shown in V_{X-1} but not shown in V_X in dialog "What's new".
T2	1. Click on toolbar icon "new" to create a new file.	1	A new file is created and shown in a new window in V_{X-1}, but nothing is shown in V_X.
T3	1. Click on menu "Help"; 2. Click on menu item "About Jmol".	2	The logo of Jmol is shown in V_{X-1} but not shown in V_X in dialog "About Jmol".
T4	1. Click on any widget in the main window.	1	The main window title is "Jmol" in V_{X-1}, but is some random string in V_X.

Table 5.7: JabRef Bugs

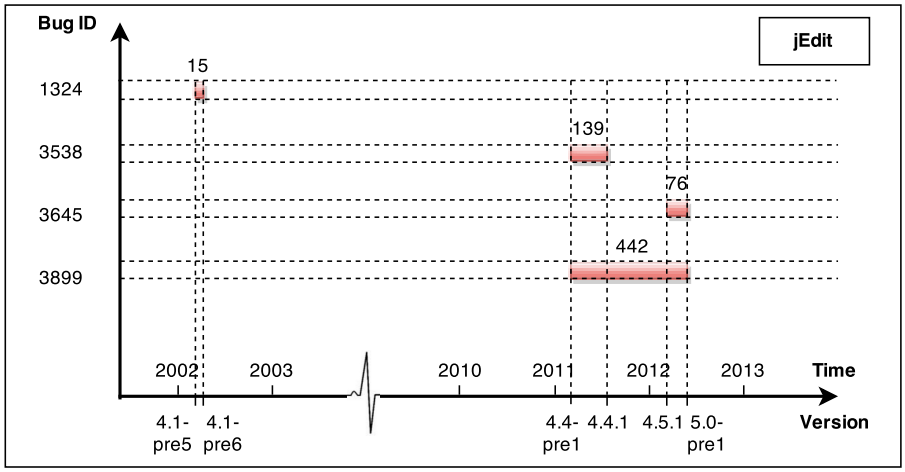
Bug Id	Steps	TC Length	Oracle
65	1. Click on menu “Options”; 2. Click on menu item “Customize entry types”; 3.Click on the help icon in the new window.	3	The widget “JabRef Help” contains help content in V_{X-1} whereas is empty in V_X.
160	1. Click on toolbar icon “new” to create a new database.	1	The search column is not wide enough nor expandable in V_X.
1130	1. Click on toolbar icon “new” to create a new database.	1	The “Enabled” property of the menu item “Close database” has a value true in V_{X-1} but a value false in V_X.
1132	1. Click on toolbar icon “new” to create a new database.	1	The “Enabled” property of the menu item “Save database” has a value true in V_{X-1} but a value false in V_X.
1133	1. Click on toolbar icon “new” to create a new database.	1	The “Enabled” property of the menu item “Search” has a value true in V_{X-1} but a value false in V_X.
1134	1. Click on toolbar icon “new” to create a new database.	1	The “Enabled” property of the menu item “Export” has a value true in V_{X-1} but a value false in V_X.
1135	1. Click on toolbar icon “new” to create a new database.	1	The “Enabled” property of the menu item “Edit Entry” has a value true in V_{X-1} but a value false in V_X.
1136	1. Click on toolbar icon “new” to create a new database.	1	The search panel on the left side of the main window is shown in V_{X-1} but missing in V_X.

Figure 5.5 visually shows the *window of exposure* (δ) of each of our bugs under consideration. We define δ as the time for which a *regression* bug has existed in the software, i.e., the time that elapsed between V_X and V_{X+1}. The x-axis shows clock time, in years, increasing from left to right, and the corresponding version numbers. The y-axis shows the individual bugs. Each bug is represented by a horizontal bar that starts at the time the bug was first introduced (not detected) into the software and ends when the bug was removed. The length of the bar indicates the window of exposure. We see that our selected bugs' window of exposure varies across applications and bugs. Most bugs in JabRef and jEdit persisted for months whereas most from Jmol persisted across years. For example, Bugs T3 and T4 of Jmol, and Bug 113x of JabRef persisted across major versions. Other bugs only persisted across multiple minor versions. Bug T4 of Jmol remained unresolved by the writing of this paper, and we denote its V_{X+1} as NA.

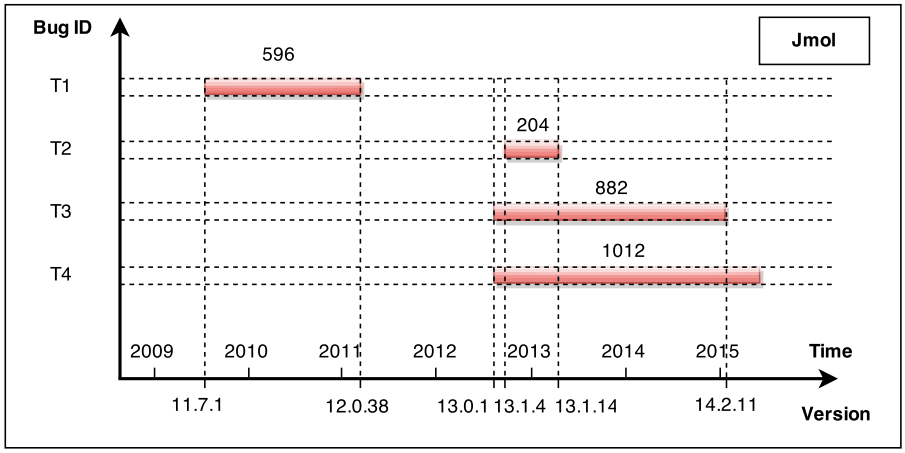
5.2.3 Addressing RQ4

In order to compute our flakiness score, *RQ4* required that we execute each test case multiple times on their respective SUTs. We executed each test case 10 times; this is consistent with the number of runs that other researchers have used in related reported work [35]. Table 5.8 shows the number of flaky tests, objects, and assertions we encountered.

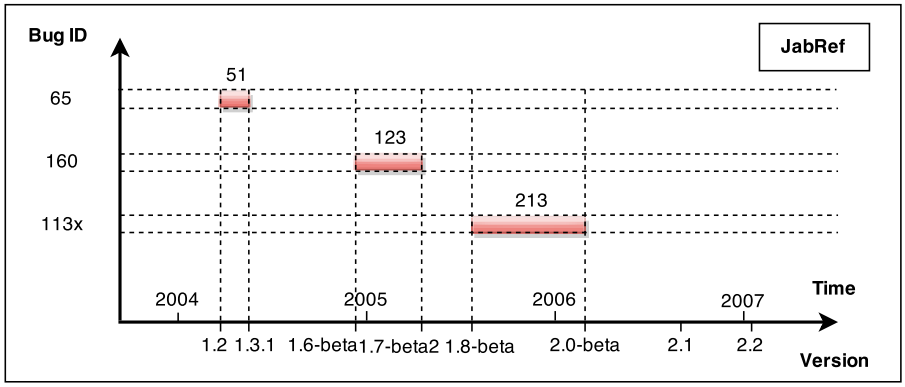
The two columns under "Flaky Objects" are for objects whose properties are flaky: Column "#" shows the number of flaky objects and Column "%" shows



(a)



(b)



(c)

Figure 5.5: Window of Exposure for Bugs in Our Study

Table 5.8: Flaky Tests, Objects and Assertions

SUT	Version	Flaky Tests		Flaky Objects		Flaky Assertions	
		#	%	#	%	#	%
jEdit	4.1Pre4	100	**100.00**	339	37.33	1590	4.79
	4.3.3	73	**100.00**	234	20.78	1064	2.84
	4.4.2	42	**100.00**	238	19.83	1001	3.89
Jmol	11.6.27	16	**100.00**	112	22.49	386	3.77
	12.2.34	26	74.29	115	10.61	288	1.03
	13.1.3	19	**100.00**	127	26.96	518	4.08
JabRef	1.1	41	**100.00**	89	36.03	296	4.72
	1.5	54	54.00	157	33.26	937	3.93
	1.7.1	95	95.00	390	**62.60**	1533	**7.19**
Total	All	466	**88.59**	1801	**27.17**	7613	**3.83**

its percentage in all objects. Similarly, the two columns under "Flaky Assertions" show the number of flaky assertions and its percentage in all assertions. We can see from the table that flaky assertions constitutes up to 7.19% of all assertions and 3.83% of the assertions are flaky overall. Although the percentage appears small, flaky assertions are observed on up to 62.60% objects and 27.17% of the objects are impacted by flaky assertions overall.

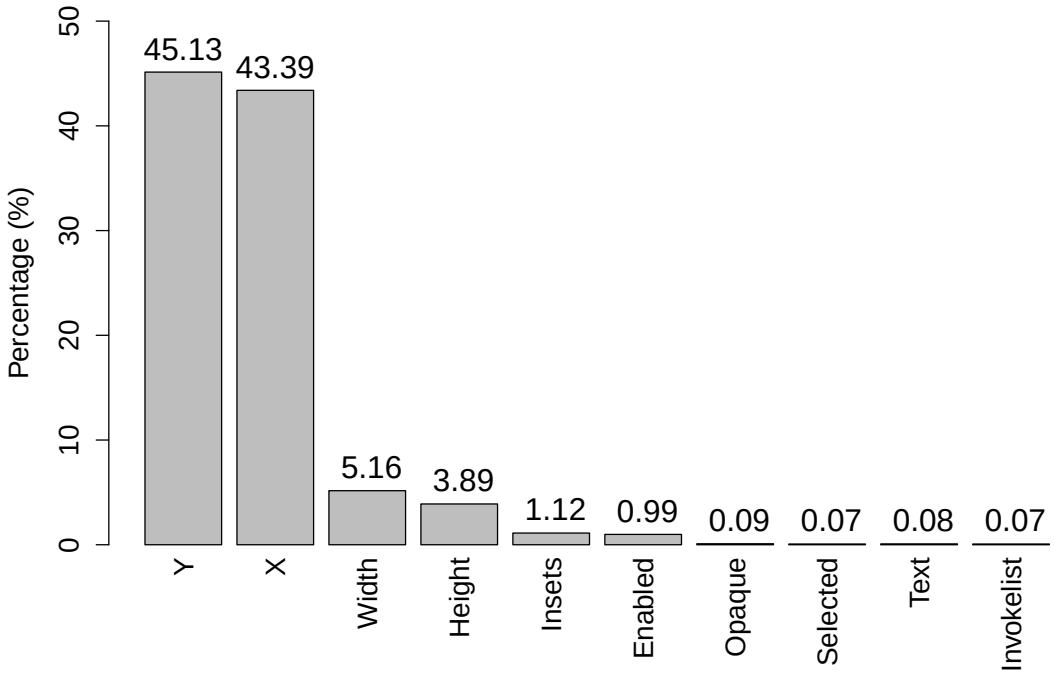


Figure 5.6: Compositions of Flaky Assertions

We further studied the compositions of the flaky assertions in terms of object properties on which these assertions were based. In Figure 5.6, the most common properties leading to flaky assertions are listed and sorted by their percentage. In the column chart, we also mark the percentages of the most dominant flaky properties, including "X" , "Y" , "Width", "Height", and "Insets". Some other UI-related prop-

erties like “Opaque”, and function-related properties like “Enabled”, “Selected”, “Text” and “Invokelist” (the list of windows opened by the widget). Because of the nature of the test cases and their assertions, all the objects are related to the GUI layer.

5.2.4 Addressing RQ5

To detect regression bugs, we want test cases to be executed on *two consecutive versions* V_{X-1} and V_X. To do this, we had to pick a subset of our bug-revealing test cases that can be run on both V_{X-1} and V_X; the number of test cases is shown in Table 5.9. To create our flake filter, we start with bug-revealing test cases for V_{X-1}. Then we filter out tests that are not executable on V_X. For each of the remaining tests, we execute them on both V_{X-1} and V_X 10 times. The final picked bug-revealing test cases that are executed on both versions.

5.2.4.1 Composition of Failures

As expected, some tests and assertions that passed on V_{X-1} fail on V_X. Those are reported as regression failures. Table 5.10 lists the average number of assertions and reported failures for each bug over all the bug-revealing test cases. Column “Assertions” shows the number of assertions. The two columns under “Failures” present the two types of failures based on the test oracles – “Objects” stands for the number of objects that no longer exists on V_X and “Assertions” stands for the failed assertions. The table shows that an average bug-revealing test case includes 2-6

Table 5.9: Test Case Selection

AUT	Bug Id	V_{X-1}	V_X	TC_{X-1}	TC_X
jEdit	1324	4.1-pre4	4.1-pre5	100	100
	3538	4.3.3	4.4-pre1	31	11
	3645	4.4.2	4.5.1	42	42
	3899	4.3.3	4.4-pre1	42	22
Jmol	T1	11.6.27	11.7.1	16	15
	T2	13.1.3	13.1.4	19	19
	T3	12.2.34	13.0.1	3	1
	T4	12.2.34	13.0.1	32	1
JabRef	65	1.1	1.2	41	19
	160	1.5	1.6-beta	100	100
	113x	1.7.1	1.8-beta	100	100

thousand assertions among which 2.42% are reported as failures. The last column presents the *true failures*, i.e., failures that reveal a bug. We obtain the numbers of true failures by manually inspecting the list of reported failures. The results show that all bugs can be detected by our regression testing technique except Bug 160 JabRef due to test harness implementation and the details are explained in the appended notes for the table.

The failures can be further categorized into 3 classes:

(1) *true failures*, i.e., failures due to a bug introduced in V_X. These failures should be presented to developer to identify and fix bugs.

(2) *update failures*, i.e., failures due to feature update in V_X. These failures should be presented to developer to update tests and assertions to match updated features of SUT.

(3) *flaky failure*. These failures are unwanted and should be filtered out properly.

5.2.4.2 Filtering Flaky Failures

To address RQ2, we first study the percentage of flaky failures among all reported failures. For each execution of a test case, we obtain the percentage of flaky failures. Then we get an average percentage value of the 10 runs. This shows the expected percentage of flaky failures when a test is executed just once. Then for each bug, we obtain a set of percentage values, one for each bug-revealing test case. The results are shown in Figure 5.7 where each box presents the distribution of percentage values of test cases revealing a bug.

Table 5.10: Regression Failures

AUT	Bug Id	Assertions	Failures		Fail Rate	True Failures
			Objects	Assertions		
jEdit	1324	6140	17	88	1.72	2
	3538	5516	19	39	1.05	4^{+}
	3645	5763	12	72	1.46	4
	3899	5706	19	52	1.26	3
Jmol	T1	2842	5	20	0.87	1
	T2	2730	4	23	0.99	1
	T3	2905	4	3	0.24	1
	T4	3063	3	0	0.10	4
JabRef	65	2687	34	204	8.86	3
	160	4241	13	403	9.80	$0^{\dagger}$
	113x	4978	17	78	1.90	1
Total	All	46571	147	982	2.42	23

+ The bug is revealed by added objects. Regression testing can detect it if new assertions are added after executing V_X.

† Detecting the bug requires verifying the "expandable" property of a GUI object.

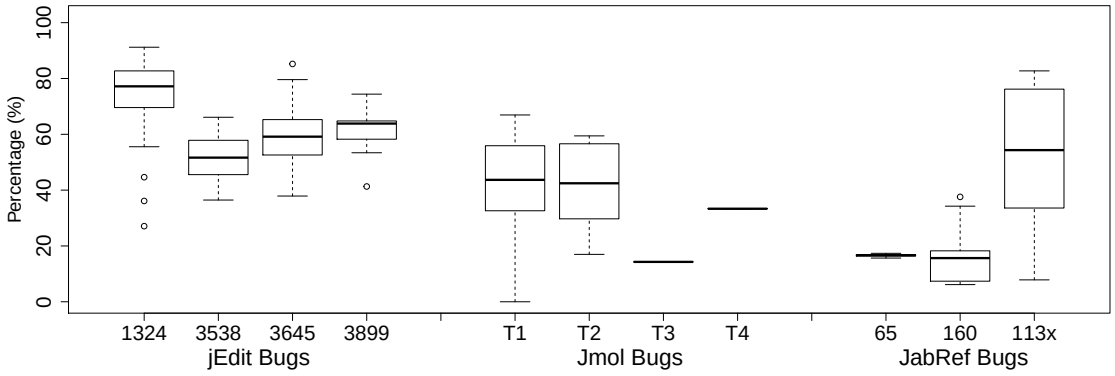


Figure 5.7: Percentage of Flaky Failures (Assertions)

From the box plot, we see that for most of the bugs, flaky failures contribute over 40% of all reported failures. For some bug-revealing test cases of Bug 1324 on jEdit, up to 91% of failures reported are due to flakiness. *It is extremely interesting to see that although only 3.83% of the assertions are flaky, they contribute to a much greater portion of reported failures. That is also one of the major reason why the test flakiness problem is important and its effects need to be minimized.*

The results show that a large fraction of failures are due to flakes, and hence should be removed before presenting the list of failures to testers or developers. By tuning the threshold for flakiness score, our flake filter can remove part or all of the flaky tests/assertions while retaining their bug-detection ability.

Next we evaluate how the thresholds impact the performance of our flake filter in terms of percentage of flaky tests eliminated vs. the percentage of bug-revealing tests retained. Figure 5.8 shows the results when different thresholds are applied to filter out flaky tests. The dash line with hollow-circle nodes are percentage of flaky tests eliminated - as the threshold increases, more flaky tests are retained,

thus the precision goes down. The solid line with filled-circle nodes are percentage of bug-revealing tests remained. The two lines are symmetric - the percentages of eliminated flaky tests and remaining bug-revealing tests always sum up to 1. This is because our experiment studied only bug-revealing test cases, all of which are flaky for the 3 bugs illustrated.

Note that in the results shown in Figure 5.8, the curves have a very steep slope at some point - meaning the flakiness score of many tests are very similar. To show more fine-grained results, we perform the same study on assertions and the results are shown in Figure 5.9. The decreasing lines show the percentage of flaky assertions eliminated when the thresholds increases; the increasing lines show the percentage of bug-revealing assertions remained along the process. We can see the dash line gradually goes down as assertions are gradually eliminated when the threshold increases. On the other hand, the percentage of bug-revealing assertions goes up as the threshold increases.

The starting point for Bug 1324 of jEdit and Bug 65 is around 50% and 80%, meaning that even with the most restricted threshold (i.e., when threshold is 0 and all flaky assertions are filtered out), some bug-revealing assertions are kept so that the bugs can still be detected. Bug T1 of Jmol is the only bug among all studied bugs that will be missed with a restricted threshold. This is because the bug is related to an image icon in a dialog whose content is encoded as part of a long HTML code that tends to have different values in different runs. Even with this bug, if we pick a higher threshold, we can detect the bug and filter out some flaky assertions in the meanwhile. The intersection of two lines shows the balance point of

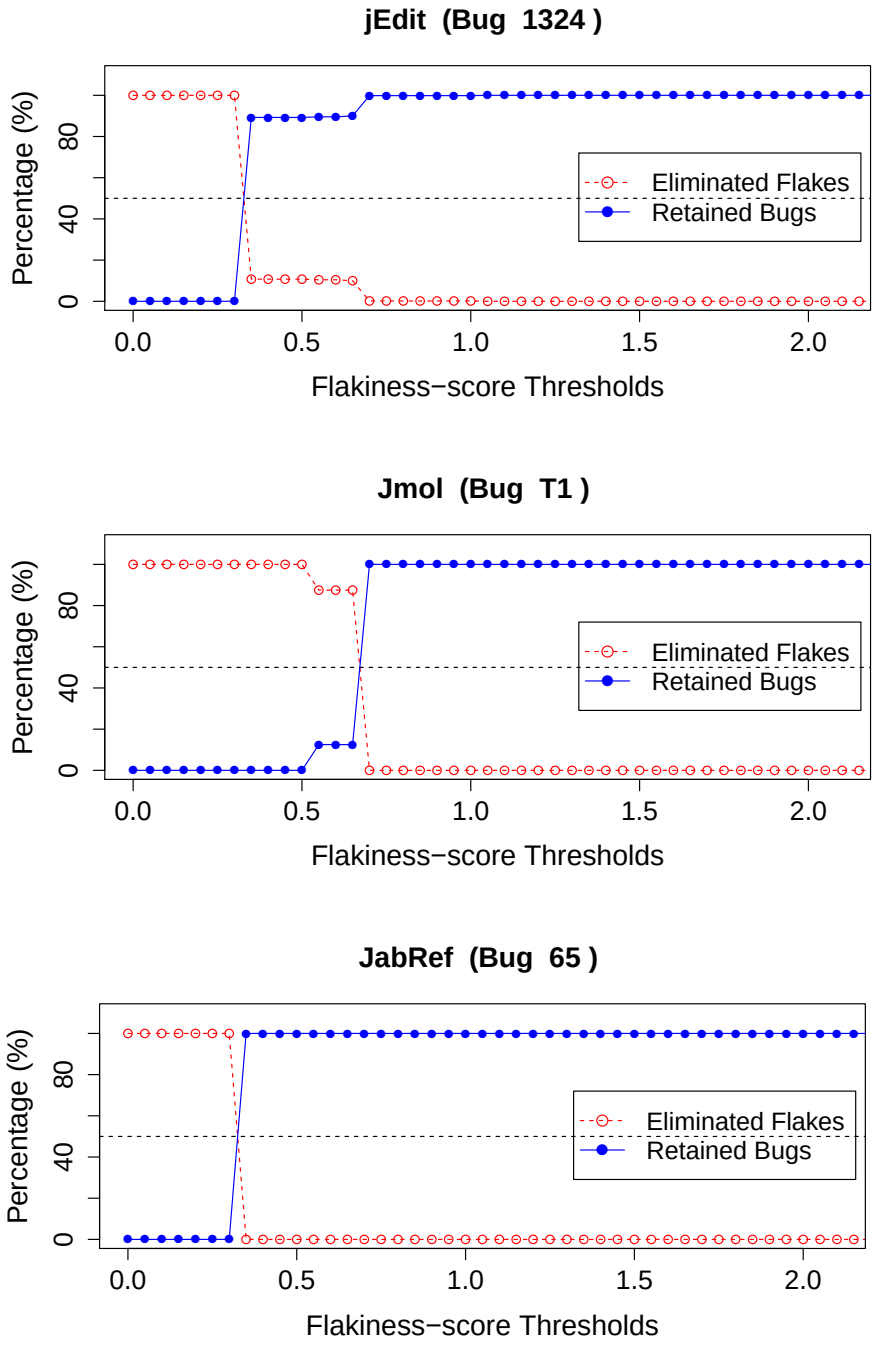


Figure 5.8: Performance of Filtering Flaky Tests using Different Thresholds

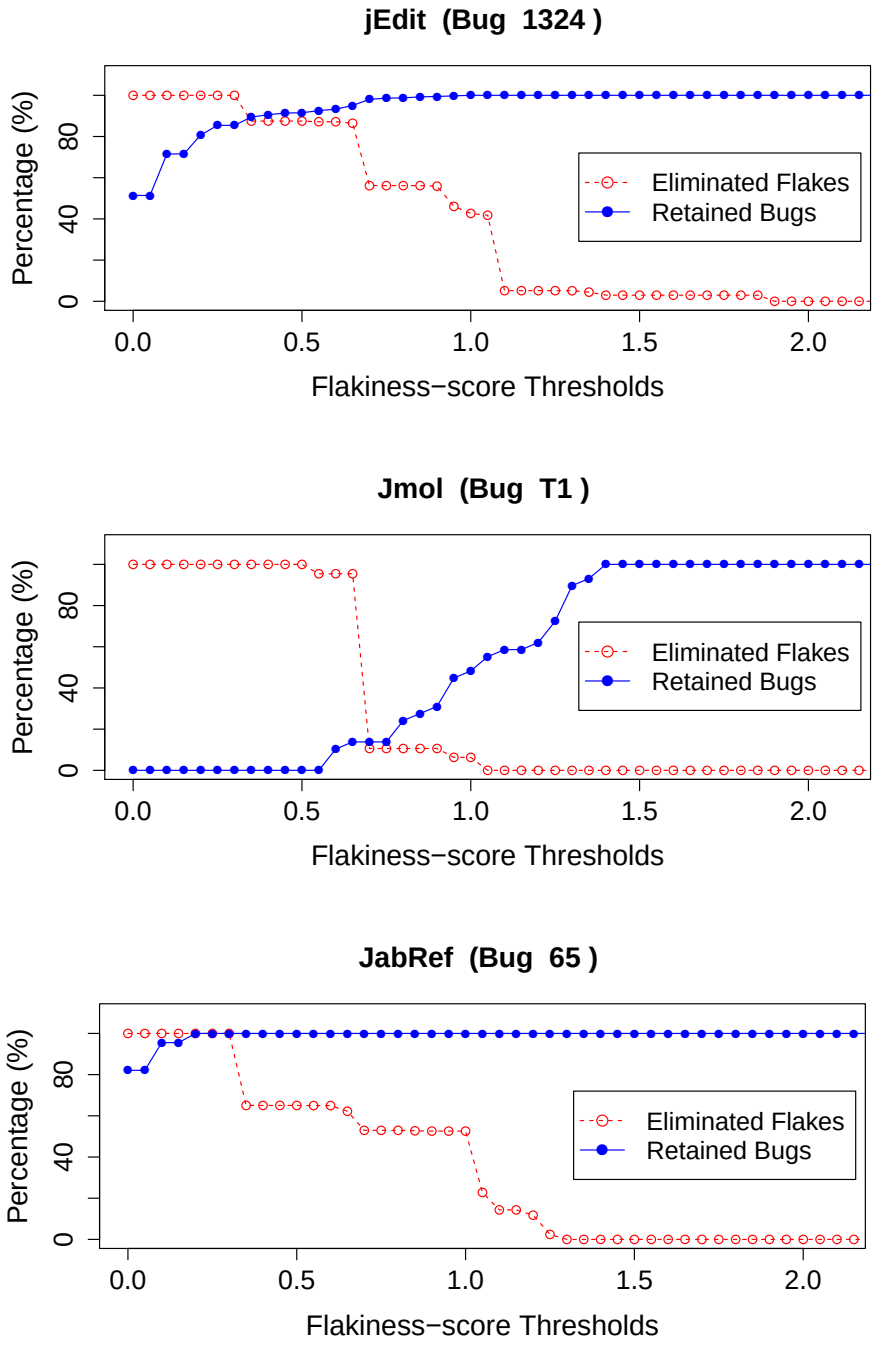


Figure 5.9: Performance of Filtering Flaky Assertions using Different Thresholds

Table 5.11: 10 Versions Studied for each Subject Application

Application	V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_8	V_9	V_{10}
JEdit	4.3.3	4.4.2	4.5.0	4.5.1	4.5.2	5.0.0	5.1.0	5.2.0	5.3.0	5.4.0
Jmol	12.2.23	12.2.24	12.2.25	12.2.26	12.2.27	12.2.28	12.2.30	12.2.32	12.2.33	12.2.34
JabRef	2.7	2.7.1	2.7.2	2.8	2.8.1	2.9	2.9.1	2.9.2	2.10-beta	2.10

our filter: at that point, the percentage of eliminated flaky assertions and remaining bug-revealing assertions are the same. For example, for Bug 1324 of jEdit, when we set the threshold to 0.3, around 85% of flaky assertions will be removed and the same percentage of bug revealing assertions will be kept. For Bug 65 of JabRef, if we pick a threshold around 0.25, our filter can remove almost all flaky assertions and keep all bug revealing assertions at the same time.

5.2.5 Addressing RQ6

To address RQ6, we run test cases on the most recent *10 consecutive versions* of the 3 subject applications that work with our tool to accumulate flaky widget-properties and evaluate its performance in removing reported failures. The versions studied are shown in Table 5.11. We will refer to these versions as V_1, V_2, ..., and V_{10}.

For each application, we first selected 100 test cases that can be executed on all its 10 versions. We executed each test case 10 times on each of the 10 versions of each application. For each version, we obtained a set of flaky assertions, and accumulated them across 10 versions. Table 5.12 shows the results on the 3 applications. For each application, we show the results in 4 rows. Row "Accu Flakes" shows the numbers

Table 5.12: Flaky Widget-Properties and Reported Failures Across 10 Versions

AUT	Measure	V_2	V_3	V_4	V_5	V_6	V_7	V_8	V_9	V_{10}
jEdit	Accu Flakes	617	1024	1119	1380	1508	1781	2165	2219	2272
	Single Flakes	617	813	484	967	988	999	1306	1321	1179
	Accu F Failures	415	659	706	833	662	812	1321	1224	1224
	Single F Failures	415	658	430	674	592	734	1258	1187	1062
	Improve(%)	0	0	**39.09**	19.09	10.57	9.61	4.77	3.02	13.24
Jmol	Accu Flakes	602	730	869	912	1020	1038	1043	1046	1089
	Single Flakes	602	547	662	841	831	795	848	937	934
	Accu F Failures	465	564	671	789	876	826	936	947	938
	Single F Failures	465	527	642	763	807	757	831	920	875
	Improve(%)	0	6.56	4.32	3.30	7.88	8.35	**11.22**	2.85	6.72
JabRef	Accu Flakes	1192	1451	1630	2182	2211	2510	2514	2628	3077
	Single Flakes	1192	1328	1003	1636	584	1243	1143	1318	1163
	Accu F Failures	1116	1342	1136	1703	1045	1378	1134	809	1144
	Single F Failures	1116	1272	709	1610	474	1175	1058	796	1100
	Improve(%)	0	5.22	37.59	5.46	**54.64**	14.73	7.44	1.61	3.85

of flaky assertions accumulated from V_1 to V_{10}.

For each pair of adjacent versions, V_{X-1} and V_X, we obtain a set of failures, i.e., assertions that passed on V_{X-1} but failed on V_X. Then we apply our flake filter to eliminate flaky failures. Rows "Accu F Failures" and "Single F Failures" show the number of failures eliminated by filtering out flaky assertions accumulated from multiple versions $(V_1, V_2, \ldots, V_{X-1})$ or obtained from a single version (V_{X-1}). The results show that up to 54.64% more flaky failures can be removed by accumulating flaky assertions along the versions.

Figure 5.10 provides a more illustrative perspective of the difference between using accumulated and single-version flaky assertions. The top line with circle nodes show the numbers of failures reported. The dash lines with hollow triangle nodes show the numbers after removing flaky failures on the single version. The solid lines with solid triangle nodes show the numbers after accumulating flaky assertions across versions and applying our filter.

5.3 Summary

This chapter studies the impact of flaky failures on test subjects with real bugs. A new flake filter is developed to eliminate flaky failures while retaining bug-revealing ones. Adaptable thresholds of flakiness score are used to balance the two goals. Experimental studies show that, in some cases, it is possible to completely eliminate flakiness without compromising fault-detection ability. And our flake filter can yield better results if evolved over successive SUT versions.

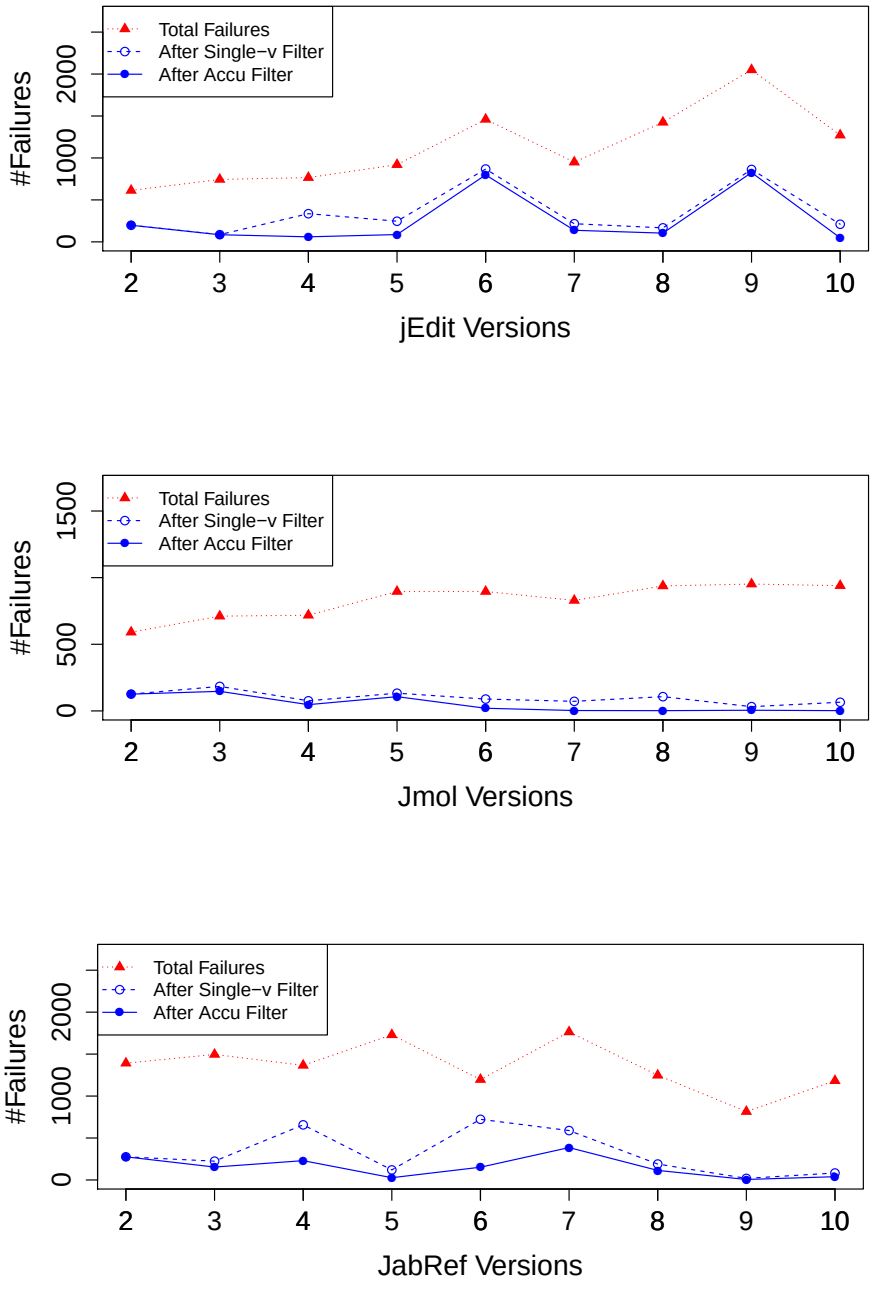


Figure 5.10: Accumulated Flaky Assertions and Ratios in Mismatches across 10 Versions

Chapter 6: Conclusions and Future Research Directions

6.1 Conclusions

As test infrastructures have evolved to handle large numbers of test cases, test harnesses have become more sophisticated to handle multiple input modalities of today's software, and test cases themselves have become more sophisticated, relying on external resources to test complex workflows, the issues surrounding test flakiness have started to take center stage [32, 33, 62]. This thesis presented a systematic approach towards the test flakiness problem.

First, an entropy-based metric was developed to quantify test flakiness. By running a test on the same SUT multiple times and observing test results at a certain output layer (e.g., code coverage, invariants, GUI state), our metric can determine the flakiness score of the test, varying from 0, indicating the test is not flaky, to a certain max value given the number of runs (e.g., 2.3 when tests are run 10 times), indicating a test is most flaky. The metric lays the foundation for our study on factors that impact flakiness and for development of the flake filter.

Second, we identified a set of factors that impact test flakiness and performed a large study to evaluate their impacts. We observed test outputs at the aforementioned three output layers, and studied factors including test execution platforms,

application starting state or configuration, test harness factors, and execution application versions. We observed as little as 0 flakiness when the identified factors are best controlled, but had flakiness as high as 2.3 (where outputs of each test run differed) when we did not control these factors. The largest flakiness was at the bottom (code) and top (GUI state) layers. Despite seeing a lower entropy at the middle, or behavioral layer, we did find some cases where invariants differed between runs. Our results suggest that all results for testing (especially from the user interface) should provide exact configuration and platform details, as well as tool parameter information. We also recommend running tests more than once and providing both averages and ranges of differences, since we are unable to completely control all variation. This also motivated us to develop other techniques to minimize effects of flaky tests.

Finally, we presented a new approach to reduce the impact of flaky tests by developing a flake filter. This filter is automatically obtained by observing multiple runs of the same version of the SUT. A flakiness score is computed for each test assertions in each test case. If the score exceeds a manually specified threshold, then the assertion is no longer used to detect failures. We have implemented a system that computes the flakiness score and applies the filter during regression testing. Our second empirical study with this implementation has shown that it is possible to automatically obtain a flake filter that, in some cases, completely eliminates flakiness without compromising fault detection ability.

6.2 Threats to Validity

As is the case with all studies, results of our two empirical studies are subject to threats to validity. To minimize threats to internal validity, we relied on robust tools, such as the GUITAR framework and the Daikon tool. We also ensured that our data is correct by continuously inspecting our data collection codes and results carefully. To minimize threats to external validity, we used open-source GUI and web applications with real bugs as our subjects; we had no influence over their codes or evolution. However, we recognize that these applications do not represent the wide range of possible applications; results could be different for other application types. Also, there are some factors we cannot fully control during our experiment. For example, we could not control the time on our Redhat cluster, but try to run tests within the same day when possible.

6.3 Future Research Directions

This research lays the foundation for much future work. We now discuss possible future directions.

- Developing a classification of modern software applications based on the flakiness likelihood of their test cases.

Modern software applications include mobile applications, software for variable devices, and software that communicates over networks. Some classes of these applications will lend themselves to flakiness; others will not. Future work can ex-

pand our empirical studies to such modern software types, quantify the flakiness of their test cases, understand factors that impact their flakiness, and at the same time, rank these application classes based on how flaky their test cases turn out.

- Developing a holistic measure of flakiness based on the full set of program elements.

In this thesis, we quantified flakiness and applied the flake filter on test cases and assertions. Future research may extend our metrics and flake filter to additional test case code statements that have outcomes, not just assertions. This extension will allow the quantification of flakiness associated with "potentially flaky" non-assertion statements, such as ones that wait for an external resource. The results may help developers or testers better understand flakiness of the tests of an SUT by providing flakiness measures for more fined-grained program elements such as code statements and blocks.

- Automatically identifying root causes of flakiness using static and dynamic analyses.

This thesis studied the general factors that impact test flakiness; but it remains a challenging job to identify root causes of flakiness. Future work may automatically tackle this problem using static and dynamic analysis techniques. Static analysis techniques such as control flow analysis [72], data flow analysis [73], and symbolic execution [74] may help understand the program elements such as variables, statements, blocks, functions that lead to flakiness. A dynamic approach may also help locate causes of flakiness by analyzing execution traces of flaky (and non-flaky) tests.

- Developing automatic classifiers for flaky tests based on machine learning techniques.

In addition to traditional program analysis techniques, future research may explore the use of supervised or unsupervised learning techniques to automatically classify flaky tests. For example, a simple binary classifier may be developed to determine if a test is flaky or not. Further, multi-class classifiers may be developed to classify flaky tests by their root causes. Features used in these classifiers may include measures used in traditional static and dynamic analysis techniques. For example, static features may include size of code covered by a test, other measures of program complexity based on programs' decision structure [75], and types of external resources the test depend on. Dynamic features may include any information collected during execution a test, such as size of log, depth of stacks of function calls, waiting time for external resources, and so on.

- Developing techniques to automatically fix flaky tests – de-flaking.

Automatically fixing flaky tests is a significant challenge. Much of the understanding that we have developed in this work may be used to de-flake tests, e.g., by avoiding assertions on certain classes of program-object properties. Manually designed strategies may be used to fix certain types of flakes. For example, flaky tests that assert on non-deterministic outputs may be fixed by using more appropriate and reliable assertions. As another example, tests that are flaky due to external resources can be fixed by using mocked objects [76].

Bibliography

[1] Qing Xie and Atif M Memon. Designing and comparing automated test oracles for gui-based software applications. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(1):4, 2007.

[2] Gnu grep 2.27, 2017.

[3] Ali Mesbah and Arie van Deursen. Invariant-based automatic testing of ajax user interfaces. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 210–220, 2009.

[4] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. Scaling regression testing to large software systems. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, SIGSOFT '04/FSE-12, pages 241–251, 2004.

[5] Aaron Marback, Hyunsook Do, and Nathan Ehresmann. An effective regression testing approach for php web applications. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ICST '12, pages 221–230, 2012.

[6] *Selenium - Web Browser Automation*, 2014 (accessed September 2014).

[7] *Mozmill - Mozilla — MDN*, 2014 (accessed September 2014).

[8] *Automation Testing Software Tools, QuickTest Professional, (Unified Functional Testing) UFT — HP Official Site*, 2014 (accessed September 2014).

[9] S. Arlt, C. Bertolini, and M. Schaf. Behind the scenes: An approach to incorporate context in GUI test case http://comet.unl.edu/generation. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, 2011.

[10] Svetoslav Ganov, Chip Killmar, Sarfraz Khurshid, and Dewayne E. Perry. Event listener analysis and symbolic execution for testing GUI applications.

In Karin Breitman and Ana Cavalcanti, editors, *Formal Methods and Software Engineering*, volume 5885 of *Lecture Notes in Computer Science*, pages 69–87. 2009.

[11] Chin-Yu Huang, Jun-Ru Chang, and Yung-Hsin Chang. Design and analysis of GUI test-case prioritization using weight-based methods. *Journal of Systems and Software*, 83(4):646 – 659, 2010.

[12] B Uma Maheswari and S Valli. Algorithms for the detection of defects in GUI applications. *Journal of Computer Science*, 7(9), 2011.

[13] Alessandro Marchetto, Filippo Ricca, and Paolo Tonella. An empirical validation of a web fault taxonomy and its usage for web testing. *Journal of Web Engineering*, 8(4):316–345, 2009.

[14] Alessandro Marchetto and Paolo Tonella. Using search-based algorithms for ajax event sequence generation during testing. *Empirical Software Engineering*, 16(1):103–140, 2011.

[15] Tommi Takala, Mika Katara, and Julian Harty. Experiences of system-level model-based gui testing of an android application. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 377–386. IEEE, 2011.

[16] Suresh Thummalapenta, K. Vasanta Lakshmi, Saurabh Sinha, Nishant Sinha, and Satish Chandra. Guided test generation for web applications. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 162–171, 2013.

[17] Zhongxing Yu, H. Hu, Chenggang Bai, Kai-Yuan Cai, and W.E. Wong. GUI software fault localization using N-gram analysis. In *High-Assurance Systems Engineering (HASE), 2011 IEEE 13th International Symposium on*, 2011.

[18] Nadia Alshahwan and Mark Harman. Augmenting test suites effectiveness by increasing output diversity. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 1345–1348, 2012.

[19] Nadia Alshahwan and Mark Harman. Coverage and fault detection of the output-uniqueness test selection criteria. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 181–192, 2014.

[20] Lijun Mei, Zhenyu Zhang, W. K. Chan, and T. H. Tse. Test case prioritization for regression testing of service-oriented business applications. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, pages 901–910, 2009.

[21] Nadia Alshahwan and Mark Harman. Automated web application testing using search based software engineering. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 3–12, 2011.

[22] Tingting Yu, W. Srisa-an, and G. Rothermel. An empirical comparison of the fault-detection capabilities of internal oracles. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, 2013.

[23] Sara Sprenkle, Emily Gibson, Sreedevi Sampath, and Lori Pollock. Automated replay and failure detection for web applications. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 253–262, 2005.

[24] Fadi Zaraket, Wes Masri, Marc Adam, Dalal Hammoud, Raghd Hamzeh, Raja Farhat, Elie Khamissi, and Joseph Noujaim. Guicop: Specification-based gui testing. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ICST '12, pages 747–751, 2012.

[25] Porfirio Tramontana, Salvatore De Carmine, Gennaro Imparato, Anna Rita Fasolino, and Domenico Amalfitano. A toolset for gui testing of android applications. In *Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM)*, ICSM '12, pages 650–653, 2012.

[26] *COMET - Community Event-based Testing*, 2014 (accessed September 2014).

[27] Si Huang, Myra B. Cohen, and Atif M. Memon. Repairing GUI test suites using a genetic algorithm. In *International Conference on Software Testing, Verification and Validation (ICST)*, pages 245–254, April 2010.

[28] Atif M. Memon and Myra B. Cohen. Automated testing of gui applications: models, tools, and controlling flakiness. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1479–1480, Piscataway, NJ, USA, 2013. IEEE Press.

[29] Android flakytest annotation. `http://goo.gl/e8PILv`. 2016-10-05.

[30] Flakiness dashboard howto. `http://goo.gl/JRZ1J8`. 2016-10-05.

[31] Pooja Gupta, Mark Ivey, and John Penix. Testing at the speed and scale of google, 2011.

[32] Get the most out of test automation at scale - listen to your data. `https://goo.gl/s5zvgC`. 2017-07-05.

[33] Meeting with scientists and engineers at vmware.

[34] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 643–653. ACM, 2014.

[35] Zebao Gao, Yalan Liang, Myra B Cohen, Atif M Memon, and Zhen Wang. Making system user interactive tests repeatable: When and what should we control? In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 55–65. IEEE, 2015.

[36] Zebao Gao, Chunrong Fang, and Atif M Memon. Pushing the limits on automation in gui regression testing. In *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*, pages 565–575. IEEE, 2015.

[37] Zebao Gao and Atif M Memon. Which of my failures are real? using relevance ranking to raise true failures to the top. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, pages 62–69. IEEE, 2015.

[38] Zebao Gao and Atif M Memon. An entropy-based automatic flake filter to reduce flakiness in testing. In *International Conference on Software Testing, Verification and Validation (ICST)*, 2018.

[39] Zebao Gao, Zhenyu Chen, Yunxiao Zou, and Atif M Memon. Sitar: Gui test script repair. *Ieee transactions on software engineering*, 42(2):170–186, 2016.

[40] Emil Alégroth, Zebao Gao, Rafael Oliveira, and Atif Memon. Conceptualization and evaluation of component-based testing unified with visual gui testing: an empirical study. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, pages 1–10. IEEE, 2015.

[41] Rafael AP Oliveira, Emil Alégroth, Zebao Gao, and Atif Memon. Definition and evaluation of mutation operators for gui-level mutation analysis. In *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*, pages 1–10. IEEE, 2015.

[42] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. Taming google-scale continuous testing. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*, pages 233–242. IEEE Press, 2017.

[43] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on software engineering*, 27(10):929–948, 2001.

[44] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *IEEE Transactions on software engineering*, 22(8):529–551, 1996.

[45] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D Ernst, and David Notkin. Empirically revisiting the test independence assumption. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 385–396. ACM, 2014.

[46] Jonathan Bell and Gail Kaiser. Unit test virtualization with vmvm. In *Proceedings of the 36th International Conference on Software Engineering*, pages 550–561. ACM, 2014.

[47] Tim Miller et al. Using dependency structures for prioritization of functional test suites. *IEEE transactions on software engineering*, 39(2):258–275, 2013.

[48] Kivanç Muşlu, Bilge Soran, and Jochen Wuttke. Finding bugs by isolating unit tests. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 496–499. ACM, 2011.

[49] Eitan Farchi, Yarden Nir, and Shmuel Ur. Concurrent bug patterns and how to test them. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 7–pp. IEEE, 2003.

[50] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ACM Sigplan Notices*, volume 43, pages 329–339. ACM, 2008.

[51] Scott D Stoller. Testing concurrent java programs using randomized scheduling. *Electronic Notes in Theoretical Computer Science*, 70(4):142–157, 2002.

[52] Alan Hartman, Andrei Kirshin, Kenneth Nagin, Sergey Olvovsky, and Aviad Zlotnick. Model based test generation for validation of parallel and concurrent software, August 8 2006. US Patent 7,089,534.

[53] Jun Yan, Zhongjie Li, Yuan Yuan, Wei Sun, and Jian Zhang. Bpel4ws unit testing: Test case generation using a concurrent path analysis approach. In *2006 17th International Symposium on Software Reliability Engineering*, pages 75–84. IEEE, 2006.

[54] William Pugh and Nathaniel Ayewah. Unit testing concurrent software. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 513–516. ACM, 2007.

[55] Selenium. `http://www.seleniumhq.org/`. 2017-01-10.

[56] Bao Nguyen, Bryan Robbins, Ishan Banerjee, and Atif Memon. GUITAR: an innovative tool for automated testing of gui-driven software. *Automated Software Engineering*, pages 1–41, 2013.

[57] Andrea Arcuri, Gordon Fraser, and Juan Pablo Galeotti. Automated unit test generation for classes with environment dependencies. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 79–90. ACM, 2014.

[58] Xun Yuan, Myra Cohen, and Atif M. Memon. Covering array sampling of input event sequences for automated GUI testing. In *International Conference on Automated Software Engineering*, pages 405–408, 2007.

[59] Zhen Ming Jiang, Ahmed E Hassan, Gilbert Hamann, and Parminder Flora. Automated performance analysis of load tests. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 125–134. IEEE, 2009.

[60] B Dasarathy. Timing constraints of real-time systems: Constructs for expressing them, methods of validating them. *IEEE transactions on Software Engineering*, (1):80–86, 1985.

[61] BM Subraya and SV Subrahmanya. Object driven performance testing of web applications. In *Quality Software, 2000. Proceedings. First Asia-Pacific Conference on*, pages 17–26. IEEE, 2000.

[62] John Micco and Atif Memon. Gtac2016: How flaky tests in continuous integration: Current practice at google and future directions.

[63] Atif M Memon, Ishan Banerjee, and Adithya Nagarajan. Gui ripping: Reverse engineering of graphical user interfaces for testing. In *Proceedings of The 10th Working Conference on Reverse Engineering*, volume 3, page 260, 2003.

[64] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, December 2007.

[65] Kelly Androutsopoulos, David Clark, Haitao Dan, Robert M. Hierons, and Mark Harman. An analysis of the relationship between conditional entropy and failed error propagation in software testing. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 573–583, 2014.

[66] J. Campos, R. Abreu, G. Fraser, and M. d'Amorim. Entropy-based test generation for improved fault localization. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, 2013.

[67] Shin Yoo, Mark Harman, and David Clark. Fault localization prioritization: Comparing information-theoretic and coverage-based approaches. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(3):19, 2013.

[68] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[69] GUITAR – a GUI Testing frAmewoRk. website, 2009. http://guitar.sourceforge.net.

[70] *Cobertura*, 2014 (accessed September 2014).

[71] Yunxiao Zou, Zhenyu Chen, Yunhui Zheng, Xiangyu Zhang, and Zebao Gao. Virtual dom coverage for effective testing of dynamic web applications. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 60–70. ACM, 2014.

[72] Frances E Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM, 1970.

[73] Frances E. Allen and John Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137, 1976.

[74] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[75] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[76] Nikolai Tillmann and Wolfram Schulte. Mock-object generation with behavior. In *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*, pages 365–368. IEEE, 2006.