

TOWARDS EFFECTIVE TEST ORACLE AUTOMATION

by

Kesina Baral  
A Dissertation  
Submitted to the  
Graduate Faculty  
of  
George Mason University  
in partial fulfillment of  
The Requirements for the Degree  
of  
Doctor of Philosophy  
Computer Science

Committee:

\_\_\_\_\_ Dr. Jeff Offutt, Dissertation Director  
\_\_\_\_\_ Dr. Paul Ammann, Committee Member  
\_\_\_\_\_ Dr. Kevin Moran, Committee Member  
\_\_\_\_\_ Dr. Vivian Motti, Committee Member  
\_\_\_\_\_ Dr. David Rosenblum, Department Chair

Date: \_\_\_\_\_ Fall Semester 2022  
George Mason University  
Fairfax, VA

Towards Effective Test Oracle Automation

A dissertation submitted in partial fulfillment of the requirements for the degree of  
Doctor of Philosophy at George Mason University

By

Kesina Baral  
Master of Science  
George Mason University, 2021  
Bachelor of Engineering  
Tribhuwan University, 2015

Director: Dr. Jeff Offutt, Professor  
Department of Computer Science

Fall Semester 2022  
George Mason University  
Fairfax, VA

Copyright © 2022 by Kesina Baral  
All Rights Reserved

# Table of Contents

	Page
List of Tables . . . . .	vi
List of Figures . . . . .	vii
Abstract . . . . .	viii
1 Introduction . . . . .	1
1.1 Overview . . . . .	1
1.2 Challenges of Test Automation . . . . .	2
1.3 Problem Statement and Motivation . . . . .	7
1.4 Thesis Statement and Contribution . . . . .	9
1.5 Outline of the Dissertation . . . . .	10
2 Background . . . . .	12
2.1 Testing and Testability . . . . .	12
2.2 Test Automation . . . . .	14
2.3 The RIPR model . . . . .	15
2.4 Test Oracles . . . . .	17
2.5 Test Oracle Problem . . . . .	17
2.6 Mobile Devices and Graphical User Interface Testing . . . . .	19
2.6.1 Challenges of GUI testing . . . . .	21
2.6.2 Android Development Tools and Frameworks . . . . .	22
2.6.2.1 Android Virtual Devices . . . . .	22
2.6.2.2 Android Debug Bridge . . . . .	22
2.6.2.3 UI Automator . . . . .	22
3 Related Work . . . . .	24
3.1 Test Oracle Problem . . . . .	24
3.2 Related Work to Smart Tests . . . . .	26
3.3 Automated Mobile Testing . . . . .	27
4 Test Oracle Problem - Blind Tests . . . . .	29
4.1 Challenges of Creating Test Oracles . . . . .	31
4.2 Empirical Evaluations of Test Oracles . . . . .	32
4.2.1 Methodology . . . . .	32


4.2.2	Study 1: Preliminary Analysis . . . . .	33
4.2.3	Study 2: Students in a Fourth Year Testing Class . . . . .	34
4.2.3.1	Subject and Program Selection . . . . .	34
4.2.3.2	Program Requirements . . . . .	35
4.2.3.3	Known Faults . . . . .	36
4.2.3.4	Data Collection . . . . .	37
4.2.3.5	Test Evaluation . . . . .	37
4.2.4	Study 3: More Undergraduate Students . . . . .	38
4.2.4.1	Seeded Faults . . . . .	39
4.2.4.2	Data Collection and Test Evaluation . . . . .	39
4.2.5	Study 4: Professionals at a Software Engineering Company . . . . .	39
4.3	Observations and Results . . . . .	40
4.3.1	Root Causes of Blind Tests . . . . .	43
4.4	Threats to Validity . . . . .	46
4.5	Conclusions and Future Work . . . . .	46
5	Test Maintenance Strategy - Smart Tests . . . . .	48
5.1	Background and Challenges of Test Suite Management . . . . .	50
5.1.1	Managing Tests by Hand . . . . .	50
5.1.2	Information Needed . . . . .	51
5.1.3	Central Management vs. Test Responsibility . . . . .	52
5.1.4	Test Self-management . . . . .	52
5.2	A Framework for Test Self-management . . . . .	52
5.2.1	Test Framework's Five Step Process . . . . .	53
5.2.2	Details of the Test Framework Process . . . . .	56
5.2.3	Scope of the Framework . . . . .	61
5.3	An Empirical Study . . . . .	61
5.3.1	Methodology . . . . .	62
5.3.1.1	Study One: Preliminary Analysis . . . . .	62
5.3.1.2	Study Two: Apache Projects . . . . .	64
5.4	Observations and Results . . . . .	65
5.5	Threats to Validity . . . . .	69
5.6	Conclusions and Future Work . . . . .	69
5.6.1	Future Work . . . . .	70
6	GUI-based Test Oracle Automation - MAGNETO . . . . .	72
6.1	Introduction . . . . .	72
6.2	Behavioral Oracle Taxonomy . . . . .	75

6.2.1	Taxonomy Derivation Methodology . . . . .	75
6.2.2	Taxonomy Results . . . . .	76
6.2.3	Behavioral Oracle Taxonomy Insights . . . . .	85
6.3	MAGNETO: Automating Android Test Oracles . . . . .	86
6.3.1	Overview . . . . .	86
6.3.1.1	App Execution . . . . .	87
6.3.1.2	Trigger Detection . . . . .	87
6.3.1.3	Oracle Execution . . . . .	87
6.3.2	Oracle Implementation Details . . . . .	88
6.3.3	Oracle Descriptions . . . . .	89
6.4	Empirical Evaluation . . . . .	92
6.4.1	Study Context . . . . .	92
6.4.2	Methodology . . . . .	93
6.4.3	Evaluation Metrics . . . . .	95
6.4.4	Evaluation Results . . . . .	96
6.4.4.1	RQ 6.1: How Accurate is MAGNETO’s Oracle Trigger De- tection? . . . . .	97
6.4.4.2	RQ 6.2: What is MAGNETO’s Success Rate in Detecting Failure? . . . . .	98
6.4.4.3	RQ 6.3: How Reliable is MAGNETO in Signaling Failure? .	98
6.4.4.4	RQ 6.4: Can MAGNETO be Combined with Existing AIG Tools? . . . . .	98
6.4.4.5	RQ 6.5: How does MAGNETO Compare with Existing Work?	99
6.5	Discussion . . . . .	99
6.6	Conclusion and Future Work . . . . .	101
7	Conclusion and Future Research . . . . .	104
7.1	Research Conclusion . . . . .	104
7.2	Contributions Summary . . . . .	107
7.3	Impact . . . . .	108
7.4	Papers . . . . .	109
7.5	Future Research . . . . .	110
A	Appendix . . . . .	112
1.1	Individual Contributions to the Blind Tests Project . . . . .	113
1.2	Individual Contributions to the Smart Tests Project . . . . .	113
1.3	Individual Contributions to the MAGNETO Project . . . . .	113
	Bibliography . . . . .	139

## List of Tables

Table	Page
4.1 Study one: Tests that caused failure but did not see the failure . . . . .	33
4.2 Study two (students): Number of tests that reached each condition in the RIPR model for the five faults—137 total tests . . . . .	40
4.3 Study two (students): Frequency of propagating failures that were revealed	41
4.4 Study three (students): Number of tests that reached each condition in the RIPR model for the five faults—184 total tests . . . . .	42
4.5 Study three (students): Frequency of propagating failures that were revealed	42
4.6 Study four (professionals): Number of tests that reached each condition in the RIPR model for the five faults—14 total tests . . . . .	43
4.7 Study four (professionals): Frequency of propagating failures that were revealed	43
5.1 Study one: Subject program details of preliminary analysis . . . . .	62
5.2 Study one: PIT mutators used . . . . .	63
5.3 Study two: Subject program details . . . . .	64
5.4 Study two: Major mutation operators used . . . . .	65
5.5 Result: Study one . . . . .	66
5.6 Result: Frequency of test status match and time taken for study two . . . .	67
6.1 Oracle strategy labels . . . . .	82
6.2 MAGNETO empirical evaluation results. . . . .	96
6.3 MAGNETO results with APE inputs . . . . .	98
A.1 GUI issues and GUI sub-issues categories . . . . .	120
A.2 Complete taxonomy result . . . . .	121

## List of Figures

Figure	Page
1.1 Test automation activities . . . . .	2
1.2 An example Java source code . . . . .	5
1.3 An example JUnit test method - testConcatNames . . . . .	5
1.4 Evolved Java source code . . . . .	6
1.5 Evolved JUnit test code . . . . .	6
1.6 Flaky JUnit test code . . . . .	8
2.1 An example JUnit test method - testAdd . . . . .	15
2.2 The RIPR model (figure adapted from oracle strategies paper [1]) . . . . .	16
2.3 GUI testing idea . . . . .	20
4.1 Example Java statement and JUnit assertions . . . . .	30
4.2 Quiz retake scheduler screen . . . . .	35
4.3 Calendar output . . . . .	38
5.1 Test self-management framework . . . . .	53
5.2 Test self-management result . . . . .	60
5.3 Example code of CFG not created . . . . .	68
5.4 Example of incorrect CFG created . . . . .	68
5.5 Example four-line statement . . . . .	68
6.1 Illustration of bug #3971 from the K9Mail app . . . . .	77
6.2 Oracle taxonomy categories . . . . .	78
6.3 ( $\mathcal{L}_1$ ) Categorization of failure effects on users . . . . .	78
6.4 ( $\mathcal{L}_2$ ) Oracle analyses and failure categorization . . . . .	79
6.5 ( $\mathcal{L}_3$ ) Percentage of resources required for automated oracle . . . . .	80
6.6 ( $\mathcal{L}_4$ ) App behavior invariants derived from the taxonomy.  denotes oracle implemented invariants. . . . .	80
6.7 An overview of the workflow of MAGNETO . . . . .	86
A.1 Quiz retake scheduler web version . . . . .	112



# Abstract

TOWARDS EFFECTIVE TEST ORACLE AUTOMATION

Kesina Baral, PhD

George Mason University, 2022

Dissertation Director: Dr. Jeff Offutt

A significant change in software development over the last decade has been the growth of test automation. Automated tests that can reveal software failures not only save time and money, but also increase reproducibility, reduce errors during testing, and ultimately lead to software that is better and cheaper. However, developing and maintaining high quality test suites that can evaluate software quality and ensure reliability is challenging.

One reason for this challenge is the difficulty of creating automated test oracles. Automated tests must include code to check whether software behaves correctly on individual tests by comparing expected behavior with actual behavior, thus revealing incorrect behavior. This code is called *test oracle*. To write test oracles, developers must first understand the software requirements and identify correct behaviors. Next, developers must implement code for automated comparison on expected and actual behavior. However, the identified software behavior may evolve as the software requirements evolve, and the test suite needs to evolve as well. Managing the growing test suite by hand can result in *test bloat*, where the test suite has more tests than are needed to effectively test the software. Failure to create correct test oracles and effectively maintain test suites can result in low quality test suites and in turn low quality software.

In this research, we designed, implemented, and evaluated approaches and tools to

improve test quality by tackling the challenges of test oracles and test maintenance. First, the research presented a study of *blind tests*—a common but not widely studied test oracle problem. We conducted three empirical studies with students and professionals to identify the prevalence and root causes of *blind tests*. We found that blind tests is a serious problem for test automation and is prevalent in tests written by students as well as professionals. Second, we developed a *behavioral oracle taxonomy* for Android apps that characterizes GUI failures in different levels of abstraction. We also designed MAGNETO—an approach for automating Android test oracles. Our evaluation illustrates that MAGNETO has 94.6% accuracy on average. Finally, we also presented a framework to tackle test maintenance problem. We developed a framework to automate test management by enhancing automated tests to track software changes and re-run as needed. Our evaluation shows that the framework takes 6.87 seconds to analyze software changes and the accuracy of the framework recommended action is 94% on average.

The goal of this research is to improve software quality, through effective and efficient testing. We demonstrate that the approaches and tools presented in this research can improve software quality through a series of empirical studies and evaluations. We developed techniques and strategies to automatically create automated test oracles that are more effective than existing test oracles. We also designed techniques to effectively automate test maintenance activities that are currently done by hand.

# Chapter 1: Introduction

## 1.1 Overview

Software testing is a crucial part of software development. It is used to evaluate the trustworthiness of system units as well as the whole system by subjecting it to different scenarios. Scenarios refer to user actions and inputs provided to the system. This research focuses on unit level testing. Software testing also allows the stakeholders to assess whether a software system has satisfied its requirements. Testing has several activities such as designing the test inputs, executing the software with the test inputs, comparing the actual behavior of the software with expected behavior, and reporting the result of testing. Software testing accounts for up to 30% to 80% of software development costs [2–4].

Given the complex, repetitive, time and resource consuming, and error prone nature of the process, software engineers have been automating tests for decades. However, in the past 10 years, industry has been steadily increasing its use of test automation [5]. This is driven partly by efficiency advantages of automation [6], and partly by agile processes such as test driven development [7], which require tests to be automated. The underlying driver is, not surprisingly, economics. *Fail watch* analyzed English language news articles for a year and found 606 recorded software failures that impacted half the world’s population and cost a combined \$1.7 trillion US dollars [8]. With the rapid adoption of advanced technologies, the global test automation market is predicted to grow from USD 20.7 billion in 2021 to USD 49.9 billion by 2026 [9]. Software test automation leverages software to perform most, if not all, of the activities of software testing. It not only lowers the cost involved, it also increases the process efficiency by increasing the overall speed of testing. Furthermore, automation, by its very nature, enables repetition of tasks and reduces the risk of human error significantly. This increases effectiveness of the test process and ultimately leads to

software that is better and cheaper [10,11].

Software test automation has five important activities, as illustrated in figure 1.1.

1. Test requirement analysis: This is usually the first step in software test automation. It involves analysis of the Software Under Test (SUT) to understand what needs to be tested and to derive its test requirements.
2. Test input and test oracle design: Once the test requirements are available, the next step is to choose the inputs for the test cases, and expected behavior of the software to the test inputs.
3. Test implementation: Test values and test oracles need to be embedded in executable test scripts. This allows repeated and consistent execution of tests.
4. Test execution: The test scripts are executed on the SUT and the test result is presented to the tester.
5. Test maintenance: The SUT evolves over time and so must the test. Some changes in the SUT requires the test to be rerun as is or rerun after modification, while other changes might make the test irrelevant and the test needs to be deleted.

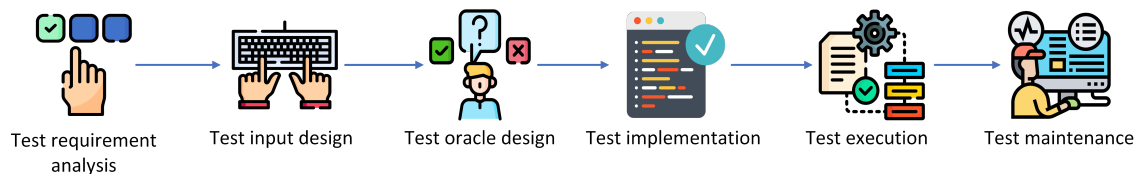


Figure 1.1: Test automation activities

## 1.2 Challenges of Test Automation

Ideally, we want to automate all the activities involved in software testing to maximize the benefits of automation. However, fundamental challenges first need to be overcome to achieve complete test automation. Here, I list some of the most relevant challenges that

researchers in test automation has been working to overcome:

### **1. Adoption of test automation practice**

Many factors make test automation challenging to incorporate during development. It is complex, it requires significant development cost, and it requires a specific skill-set from the developers. Although testing is the primary way to ensure software quality, it is often the first to be neglected or excluded altogether if time and resources are an issue [12]. With the pressure to deliver software by a deadline, imminent goals are prioritized and long-term goals like testing is neglected [13–17]. This general notion that software testing is optional and only needs to be done in ideal conditions has, by extension, also severely limited the adoption of test automation. Furthermore, some systems that are not easily testable require much more effort and expertise to test and might also include several iterations of test redesign to make it more effective. All of this contributes to making the testing process difficult, time-consuming, and expensive. Hence, the adoption of software testing in general and test automation in particular is a significant challenge.

### **2. Creating effective test cases**

An important goal of software testing is to find faults in software to assess its trustworthiness and overall quality. To find faults, we need to create effective test inputs. The faulty parts of the system have to be triggered during the program execution, leading to an observable failure. These triggers are supplied to the system in the form of test inputs. Hence, only effective tests with the right test inputs can successfully assess the quality of the system. Test inputs can be designed in two ways—human-based and criteria-based [18]. A human-based approach relies on domain knowledge of the developers and testers. Developers decide what values could be interesting for testing purposes based on their familiarity with the system. In criteria-based testing the software is represented in an abstract form such as graphs, logic expressions, or the input domain; and the test input values are selected by analyzing these abstract representations. Deciding what test inputs to use when designing tests is another challenge.

### 3. Test oracle problem

Test oracles distinguish between valid and invalid systems by comparing the actual system behavior with the expected behavior. The test oracle problem refers to the challenge of distinguishing desired system behavior from incorrect behaviors of the system under test. Consider a sorting method that sorts the student names by first name and returns the sorted order. A manual (human) oracle would have to compare the position of every name in the output list manually to test this method. This is a slow, error prone, and expensive approach. Using this approach to test the method multiple times, or with more inputs, adds to the problem.

An automated test oracle can make the behavior comparison faster, less error prone, and inexpensive than manual checking. Without an automated oracle, tests have to rely on a human to make the distinction between valid and invalid system responses. Such a dependency creates a bottleneck that inhibits automated testing from providing greater benefits with reduced costs.

However, creating automated test oracles is not a trivial task. Software behavior is often different for different test inputs, so testers need to write test-specific automated test oracles. Sometimes availability of formal specifications, *pre*-conditions and *post*-conditions, or contracts from contract driven development, can be processed to create automated test oracles.

### 4. Test adequacy

When testing a system, it is difficult yet important to answer the question: *how much testing is enough?* Testing with all the potential inputs for a program is often impossible and always impractical. Hence, testers need some way of determining when enough testing has been done. This is where formal coverage criteria come in [18]. Coverage criteria not only help in deciding the test inputs to maximize test effectiveness, but also provide a stopping rule to terminate the testing process. Determining if there is adequate testing also helps to gauge the quality and reliability of system. Several metrics such as mutation score, branch coverage, and statement coverage are derived from different test

criteria. These metrics attempt to evaluate test adequacy based on the degree of test requirement satisfied by the tests.

## 5. Test maintenance

As the requirements of the system evolve, the system under test evolves. To test this evolved system, its test suite must also evolve. This test suite evolution can require new tests to be added, unneeded tests to be removed, and the existing tests to be “evolved” in the test suite.

Consider the Java method in figure 1.2 and its JUnit test in figure 1.3. The method “concatNames” in figure 1.2 takes in two values, concatenates them with a space in the middle, and returns the resulting string.

```
public String concatNames (String x, String y) {  
    String firstName = x;  
    String lastName = y;  
    String z = firstName+' '+lastName;  
    return z;  
}
```

Figure 1.2: An example Java source code

```
@Test  
public void testConcatNames() {  
    String result = concatNames("Anita","Borg");  
    assertTrue(result.contains("Anita"));  
    assertTrue(result.contains("Borg"));  
}
```

Figure 1.3: An example JUnit test method - testConcatNames

If the source code evolves by adding a colon between the strings as shown in figure 1.4, the automated test also needs to evolve. For this example, it might be done by adding a new test or modifying an existing test. An example of an evolved automated test for

```

public String concateNames (String x, String y) {
    String firstName = x;
    String lastName = y;
    String z = firstName+':'+lastName;
    return z;
}

```

Figure 1.4: Evolved Java source code

```

@Test
public void testConcateNames() {
    String result = concateNames("Anita","Borg");
    assertTrue(result.contains("Anita"));
    assertTrue(result.contains("Borg"));
    assertTrue(result.contains(":"));
}

```

Figure 1.5: Evolved JUnit test code

the source code in figure 1.2 is shown in figure 1.5.

In general, the growth of the test suite corresponds to the growth of the system. Managing the growing test suite by hand is difficult and can result in test bloat. *Test bloat* refers to the accumulation of unnecessary tests that add no value and waste resources. Hence, finding an efficient approach to maintain tests is a challenge in software test automation.

Identifying software behavior to create automated test oracle is a non-trivial task. Furthermore, software behavior evolves over time adding to the challenge of test automation. Hence, of these five challenges, this dissertation focuses on test oracles (problem 3) and test maintenance (problem 5). The work presented in this dissertation addresses the issues by proposing and investigating new techniques to automate test automation. The goal is to significantly reduce developers' burden, allowing them to focus on critical tasks and facilitating an efficient and cost-effective process.



### 1.3 Problem Statement and Motivation

Problem Statement:

Incorrect test oracles and test bloat reduces test suite quality, which leads to low quality software.

The goal of software testing is to evaluate software quality and ensure reliability. One way to do this is by writing automated tests, particularly automated oracles. However, writing correct oracles for software is challenging. Incorrect or incomplete test oracles can lead to incorrect test results. Incorrect test results could mean that the test incorrectly reports that the software failed or the test incorrectly reports that the software passed despite software failure.

Challenges in test oracle creation stem from multiple reasons. One is the difficulty of defining correct program behavior. To write correct and complete test oracles, developers need to understand the software requirements and be able to reason about the correct software behavior. Developers face the challenges of program comprehension along with other unique challenges of navigating through code, specifications, documentations, and old tests to interpret the correct behavior.

Another challenge in test oracle creation arises from difficulties in modeling software behavior. GUI-based, event-driven software is challenging to model and hence challenging to develop test oracles for. Software that has non-deterministic output or complicated output present unique challenge of test oracles being flaky and brittle [19–21]. *test2*, shown in figure 1.6, is an example flaky test. If *test2* is run after *test1*, it passes, but if *test1* is run after *test2*, it fails. The result of *test2* depends on the test execution order.

This leads to test oracles either being manually defined in the tests or limited in their fault revealing capabilities.

However, it is not only the behavior modeling and interpretation that is difficult, but also the automation. We refer to the *automation* of a test oracle as a snippet of code that is capable of automated comparison between expected and actual software output.

The identified software behavior may evolve over time as the requirements change and the test needs to evolve as well. Some tests might need no updates, but others might need to be changed, while some others may no longer be valid and need to be deleted. This test suite management is currently done by hand, which results in test bloat. *Test bloat* is the accumulation of tests that are no longer useful, or loss of valuable tests, and waste of resources. This further hinders in developing better quality software.

```
static int value = 0;

@Test
public void test1() {
    String result = concatenateNames("Anita", "Borg");
    assertTrue(result == "Anita Borg");
    value = 1 ;
}

@Test
public void test2() {
    assertTrue(value == 1);
}
```

Figure 1.6: Flaky JUnit test code

Incorrect test oracles waste resources, take away the opportunity to improve software quality by correcting faults, and lead to poor quality software. The consequences of poor quality, unreliable software can range from inconvenience to fatalities.

The research presented in this dissertation is motivated by the challenges of test oracle automation and test maintenance. This research sheds light on issues faced by developers, and testers during test oracle automation, test maintenance and presents approaches to address them. My work focuses on two domains of software: desktop applications and mobile applications.

## 1.4 Thesis Statement and Contribution

Thesis Statement:

Effective automated test oracles and automated test maintenance can improve test suite quality, which in turn will lead to higher quality software.

To verify the thesis statement, the dissertation presents several results focusing on the test oracle problem and test suite maintenance. We conducted experimental evaluations to investigate current usage, and challenges faced by developers in creating automated test oracles. Based on the findings, we developed approaches to overcome the challenges and help developers design and implement effective and efficient test oracles.

We investigated a particular type of incorrect test oracles where a key part of the output is not observed by the oracle, resulting in *blind* tests. We conducted three empirical studies with students and professionals. This study sheds more light on the prevalence of this problem and its root causes with the following research questions:

- RQ 4.1<sup>1</sup> What percentage of tests are blind?
- RQ 4.2 Do some groups of testers write more blind tests than other groups?

Further, we designed a framework to tackle one of the causes of incorrect test oracles—misunderstood requirements and difficulties in test management. We focused on developing a framework to automate test management by making tests smarter. A test is *smart* if it can decide when it needs to be run, ignored, modified, or discarded after the software is changed, with few to no developer changes. We also conducted an empirical study to evaluate the accuracy of the analysis result from the framework—the framework suggested actions, and its execution cost with two research questions:

- RQ 5.1 How much time does the analysis use?
- RQ 5.2 How accurate are the analysis results from the framework?

After evaluating the applicability of the test management framework, we investigated test oracle automation for mobile apps. We modeled common GUI app behaviors to derive

---

<sup>1</sup>RQ I.A is tied to Chapter I

a *behavioral oracle taxonomy* and developed an automated approach, MAGNETO (autoMAT-inG aNdroid tEsT Oracles), for automatic failure detection. We then empirically evaluated five automated oracle patterns on 15 real-world faults mined from open source Android apps. Finally, we empirically evaluated MAGNETO with the following research questions:

- RQ 6.1 How accurate is MAGNETO’s oracle trigger detection?
- RQ 6.2 What is MAGNETO’s success rate in detecting failure?
- RQ 6.3 How reliable is MAGNETO in signaling failures?
- RQ 6.4 Can MAGNETO be combined with existing AIG tools?
- RQ 6.5 How does MAGNETO compare with existing work that detects UI Display Issues?

## 1.5 Outline of the Dissertation

This dissertation is organized as follows.

**Chapter 2** provides background related to testability, test automation, components of an automated test case, the Reachability, Infection, Propagation, and Revealability (RIPR) model, test oracles, the test oracle problem, and mobile software development practices and graphical user interface. It also discusses the challenges of automated mobile GUI testing.

**Chapter 3** discusses work related to (i) the test oracle problem, (ii) automated test maintenance, and (iii) automated mobile testing.

**Chapter 4** identifies and studies a common problem where test assertions are written incorrectly, causing incorrect behavior to not be recognized by the tests. We call these tests *blind* as they do not see the incorrect behavior. We describe three human-based studies and present results that assess the frequency of blind tests with different software and different user populations. The content of this chapter is based on the paper on blind tests [22].

**Chapter 5** introduces a novel and comprehensive solution to the problem of test suite management. Our approach gives individual tests the ability to self-manage through self-awareness and self-determination. We built a framework that generates control flow graphs of program modules, maps individual tests to the graph requirements for self-awareness, and does syntactic program comparison to detect software evolution and decide the appropriate

action to take after the software changes. We conducted empirical studies on open source software to evaluate the accuracy of framework recommended action and execution cost. The content of this chapter is based on the paper on smart tests [23].

**Chapter 6** presents techniques to automate the construction of GUI-based test oracles for mobile apps. We characterize common behaviors associated with failures into a behavioral oracle taxonomy. Our taxonomy identifies and categorizes common GUI element behaviors, expected app responses, and failures from a large set of reproducible bug reports. We then use the taxonomy to create a novel app-independent process to develop automated test oracles. The developed test oracles uses computer vision of user interface screens, and natural language processing of responses to detect non-crashing failures in the app. The content of this chapter is based primarily on the MAGNETO paper which is under submission.

## Chapter 2: Background

This chapter introduces theoretical and practical concepts in testing, and test automation concepts as defined in several related papers on test automation [7,18,24]. This chapter also presents background on test oracles, the test oracle problem, and graphical user interface testing.

### 2.1 Testing and Testability

Testing is an important part of software development, where software testers try to find and eliminate software problems before they are inflicted on the users. When software failures go unidentified, it can cause major inconveniences to users. For instance, an international conference I registered for asked my country of residence but did not have the United States in the option list. I had to contact the conference organizers and a five minute process took me two days to complete. Poor software quality can be extremely expensive as well. A 2020 study found that poor software quality cost \$2.08 trillion to the US economy [25]. Therefore, companies test software to ensure quality, and prevent financial and reputation loss due to software failures.

Software testing is the process of checking if software fails under some use scenarios. It involves activities such as designing scenarios to test with (test inputs), running the software with the test inputs, evaluating the software's response to those inputs (its behavior), and reporting whether the behavior was as expected. Identifying inputs, expected outputs, and valid states of the program is an important and challenging task for effective test design. This dissertation uses the terms software, application, and program interchangeably. For programs that are user event-driven like mobile apps, or embedded systems that interact

with physical environment like Google Home, it is more difficult to identify the input, output, and state details. For other programs (like desktop applications), the inputs, expected outputs, and state details are either specified in their documentation or can be inferred from the program. Such programs are said to have higher *testability*. Freedman describes testability as a property of an easily testable program [26]. He describes these properties as the ability to generate finite numbers of non-redundant sets of test cases that can easily locate faults in the program and do not have input-output inconsistencies. He further defines domain testability as the ease of program modification to test the program against intended specification using two concepts: controllability and observability.

A software component is said to be *controllable* if we can produce desired output from a specified input [26]. Controllability measures the ease of generating and providing input to a software component to force desired output. Outputs that depend on environmental factors such as humidity, temperature, or motion sensors decrease controllability.

*Observability* is the ease of observing the impact of some input on the program output [1,26]. Program outputs that are hard to observe, such as changes to large databases, message to remote computers, or signals to external hardware, have an observability problem. Although widely known in practice, observability problems have been inadequately studied by researchers. Difficulty in identifying or evaluating the impact of input leads to incorrect or incomplete tests. In my research, I have observed that even experienced programmers and testers make mistakes due to low observability.

Traditionally, test activities such as deciding and providing test inputs, test execution, and system response evaluation are manual, making the test process error prone and costly. Automating test activities allows software developers to test software with thousands of tests continuously, which increases software quality while also saving time, energy, and money.

## 2.2 Test Automation

Test cases are comprised of several pieces—test inputs, prefix values, post-fix values, expected result, and a mechanism for executing application under test. Test automation is the use of software or tools to automate these different components of a test case. Here, I focus on test inputs and test oracles.

*Test inputs* are values needed to complete an execution of the application under test [18]. Test inputs might be sequences of method calls to an object or subsystem, including all necessary objects, parameters, and resources, or user-level inputs such as text values and user interface selections. The specifics depend on the software under test, but the concepts are the same. A test is often designed with a goal or criterion, such as push all physical buttons on the mobile phone (power button, volume up button, volume down button) at least once. Test inputs are sometimes based on such test criterion that yield specific test requirements. For the test criterion mentioned above, we can derive 3 test requirements: (1) press power button, (2) press volume up button, (3) press volume down button.

In test automation, there is an additional difficulty of getting the right inputs to the desired location in the program, also known as *controllability*. When testing a Java method in isolation, most inputs are through parameters, which can be controlled directly. Consider the example JUnit test case in figure 2.1. The integers ‘2’ and ‘3’ are test inputs to the method “add()”. Thus, the method has fairly high controllability. When testing a specific *if*-statement inside a large program through its external UI, however, the tester must find input values that eventually cause the program to reach that *if*-statement, and the variables referenced in the test must have the appropriate values. In this situation, controllability is usually low, making test automation harder.

Automated tests must also include *expected results*, which are the results produced if the program satisfies its intended behavior. A *test oracle* (TO) compares the expected results with the actual results to decide if a test passes. TOs are commonly implemented as assertions in frameworks such as JUnit [27]. In the example JUnit test shown in figure 2.1, the integers ‘2’ and ‘3’ are test inputs, and the expected result is ‘5’. The test oracle



```
@Test
public void testAdd()
{
    assertTrue("testAdd incorrect", 5 == Calc.add(2, 3));
}
```

Figure 2.1: An example JUnit test method - testAdd

is the entire assertion, which directs JUnit to print the string “testAdd incorrect,” if the program result does not match the expected result.

When unit testing a method, the return value and any values printed are easy to observe. However, if the software writes to a database, controls an external sensor, or changes a shared memory object in a distributed web application, observability is harder. Many theoretical and practical challenges of test automation are due to low controllability, low observability, or both.

## 2.3 The RIPR model

Test inputs are designed in part to expose *faults*, which are static defects in the software. Given specific test inputs, a fault can result in a program *failure*, which is an external, incorrect behavior with respect to requirements or other description of expected behavior. The distinction between fault and failure led to the development of the reachability, infection, and propagation (RIP) model in the 1980s [28–31].

In past generations, testers ran tests by hand and examined the results visually. Researchers and practitioners assumed that if an error propagated to the output, the tester would notice and mark the test as failing. However, modern automated tests use assertions to compare expected with actual results automatically. Because it is often impractical and usually unnecessary to check all outputs (including external files and databases, sensors, messages, etc.), automated tests check certain specific parts of the output state. However, if the oracle does not check the part of the state that contains an erroneous value, the oracle will not see the failure. Thus, RIP was extended to include *reveal*, making the Reachability,

Infection, Propagation, and Revealability (RIPR) model [1], as illustrated in Figure 2.2.

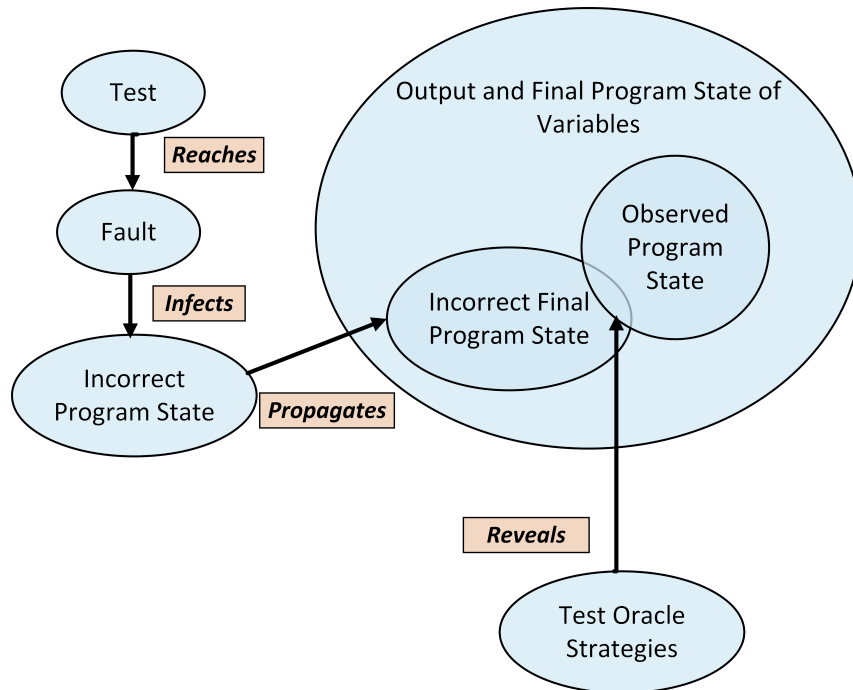


Figure 2.2: The RIPR model (figure adapted from oracle strategies paper [1])

Figure 2.2 illustrates the standard RIPR model from the testing field. Test oracles reveal faults by observing the *final program state*, that is, outputs and visible values after execution. To detect a fault, a test has to *reach* a faulty location, then the faulty code must *infect* the internal program state with incorrect values. An incorrect value must *propagate* to an **incorrect final state** (a failure). We cannot check the entire final output state, so test oracles look at part of the final state (*observed*). If the **incorrect final state** and the **observed final state** intersect, then the test reveals the failure. If they do not, then the test does not reveal the failure. That is, the test is *blind* [22]. Tests are expensive to design, create, automate, and run, effort that is largely wasted when tests cannot see failures.

## 2.4 Test Oracles

To detect software failure, we need checks in the test code that validate the software output. Such checks are called *test oracles*. As described by Richardson et al. [32], test oracles have two parts—oracle information, and oracle procedure. The *oracle information* describes the expected behavior of the application. The *oracle procedure* compares the actual behavior of the application with the expected behavior. In an automated test, a test oracle is a snippet of code that contains the oracle information and oracle procedure. An example of a test oracle is a JUnit assertion shown in figure 2.1. Test oracles are usually invoked after the application under test terminates and provides the final output, but not always. If validating intermediate application behaviors is important, the test oracles may be invoked before the application terminates. An example of this is testing mobile applications, where we may need to check states after every user event and not just the final state like the final output screen.

## 2.5 Test Oracle Problem

Test oracles are needed to distinguish desired application behavior from incorrect behaviors of the application under test. The challenge of developing a test oracle capable of making such distinction is known as the test oracle problem [33]. For some applications, we can reasonably assume the existence of test oracles, but this may not be true for all applications.

Writing correct and complete test oracle is difficult, even more so if the application to be tested is complex or non-deterministic. To write correct test oracles, developers must understand the program requirements and correct behavior. Sometimes formal specifications, pre-conditions and post-conditions, or contracts from contract driven development can be used to create and automate test oracles. In the absence of explicit program requirements, strategies such as checking for program termination, or waiting for program to crash is used as a weak test oracle [1]. Other times, test oracle automation is not possible. Without automated oracles, human testers have to make the distinction between valid and

invalid application response manually. Manual test oracles make testing slow, expensive, and error-prone.

Much test design research has focused on generating test inputs and measuring the quality of test, but relatively little research has been done to solve the test oracle problem. A comprehensive review of current literature on the test oracle problem shows four major approaches [34]:

1. Specified test oracles: Application models such as UML (Unified Modelling Language) diagrams, and state charts can be used to capture the salient behaviors of the system and later serve as specified oracles. Some of the challenges for creating specified test oracles are: (a) lack of formal specifications or application models, (b) discrepancies between the representation model and the actual application, and (c) difficulty interpreting the model output to develop test oracles.
2. Derived test oracles: This approach relies on information *derived* from different artifacts like previous software versions, documentations, software execution logs, and properties of the software under test. Derived test oracles are used to differentiate between actual and expected application behavior when specified test oracles are unavailable. Pseudo-oracles and metamorphic testing are two approaches for deriving test oracles. A pseudo-oracle is an alternate program version developed independently to determine the correct behavior. For example, output of a method written in two different languages must match and form a pseudo-oracle. Metamorphic testing relies on software properties that hold over multiple executions, called metamorphic relations. Some additional approaches of deriving test oracles are: specification mining, converting text documents into specifications, and developing regression test suites.
3. Implicit test oracles: System crashes, program termination, and error messages are some examples of implicit test oracles. Implicit oracles do not require domain knowledge and the same oracle can be used for multiple programs. However, they are not universal as some behavior like crashes might be acceptable in some contexts. Also, many failures are not so obvious.

4. Reduction of the human cost: Specified, derived, and implied oracles are useful when artifacts like specifications, documentation, and contracts exist. When such artifact are not present, testers have to manually enter inputs to the system and verify the response. This is an expensive process. However, there are some approaches, such as (a) division of testing labor through crowd sourcing, (b) employing third party companies for testing, and (c) test suite reduction, that reduce the human cost when test oracle automation is not possible.

## 2.6 Mobile Devices and Graphical User Interface Testing

Mobile devices have become pervasive in the modern society. With smartphones becoming cheaper over the years, about 50% of the world's population has become smartphone owners [35]. With Android being an open source, Linux based software used in most mobile devices, it is not surprising that Android operating system is the most popular operating system in the world. The Statcounter website reports that in February 2021, the worldwide usage share of Android OS leads that of Windows by 7.95% [36].

Part of this popularity is due to the neatly packaged versatile applications that provides information, utility, and entertainment. These mobile applications or apps are software designed to run on portable, wireless devices like tablets, smartphones, and smart watches. Some mobile applications are designed to run on a specific device and come pre-installed in the device. Other applications are available to be installed for free or at some cost from applications stores like Google Play store [37], Apple app store [38], etc.

Users interact with Android applications through its graphical user interface. Graphical User Interfaces (GUI) use visual cues like images, icons, and buttons to facilitate interactions with devices. Ease of use has made GUIs universal and they can be found in all modern devices. A GUI has several elements to provide visual stimulation (like fonts, and color), interactive design (like menus, and tabs), and intuitive information structure (like labeling, and containers). Mobile applications rely heavily on a failure free, easy to use, and engaging

GUI for its success, which makes GUI testing even more important for mobile apps [39].

As shown in figure 2.3, Graphical User Interface (GUI) testing is the process of validating the application functionality by interacting with the application as a user would, through its GUI, and observing the properties of GUI elements in the resulting GUI state. The test input of a GUI test is a valid action or a valid action sequence. A sequence of action is valid if no preceding action impedes the execution of subsequent action. A GUI test case can be defined as a valid action or a sequence of valid actions performed on a GUI state resulting in a different GUI state.



Figure 2.3: GUI testing idea

The GUI testing literature contains three main approaches: (1) manual testing, (2) record and replay, and (3) model-based testing. In manual GUI testing, a human tester performs the test activities. Manual GUI testing is particularly useful when the application changes are small and frequent, or when human expertise is crucial to distinguish whether the application passes the test. However, manual testing is time and resource consuming. With the help of test automation, mundane tasks of GUI testing can be automated for speedy result and low cost. Record and replay approach uses tools that record developer

interaction with the application and stores it as a test script. The script can be replayed later to detect faults through repeated action with different test inputs. The tester needs to manually provide the expected behavior as oracles. Model-based testing relies on abstract representation of the application through graphical models like event sequence graphs, and finite state machines. These models capture system behavior, which makes identification of valid outputs easier. Automated tests are derived from the graphical models for higher coverage.

### **2.6.1 Challenges of GUI testing**

GUI testing is riddled with challenges due its domain size, large number of possible interaction sequences, and its event driven nature. In GUI testing, every possible user action is a test case that could result in a different GUI state. Even a small application can have multitude of possible actions [40]. This creates an explosion of possible action sequences for testing, making the test process labor and time intensive.

In mobile application testing, unlike conventional software, the test oracle needs to be invoked after every user action. This is because a sequence of GUI states are generated in response to a sequence of user actions, instead of one final output state like traditional software. If an intermediate GUI state is incorrect, subsequent interactions may not be of value. Hence, the test oracles for GUI testing must verify the GUI state after every possible user interaction. Further, creating test oracles to verify every possible GUI state is a daunting task. Even if completed, the ever-changing nature of mobile applications makes it challenging to describe desired GUI element behavior in a way that is usable across applications. These challenges suggest the need to study, and recognize the behavior of common GUI elements for creating automated test oracles.

In addition to the challenges of GUI testing, mobile application testing, Android specifically, is riddled with some more challenges. Ever changing APIs, platforms, fragmented ecosystems, a competitive market, and a lack of powerful testing tools, coupled with increasing complexity of Android applications make testing arduous.

## **2.6.2 Android Development Tools and Frameworks**

This section briefly describes some of the tools and frameworks used in this dissertation.

### **2.6.2.1 Android Virtual Devices**

Android Virtual Devices (AVD) or emulators are used to simulate Android devices on desktops. Developers and testers can then run the Android applications on the emulator and interact with the application without needing the physical devices to run them on. Emulators allows various devices and Android operating system configurations to run the app. Android emulators include Nox [41], Genymotion [42], Android Studio [43], etc. I use Android Studio in my research to run Android apps on different devices and collect resources like app screenshots, screen metadata, run-time app video [43]. These resources are needed to develop a behavioral taxonomy and run automated oracles.

### **2.6.2.2 Android Debug Bridge**

Android Debug Bridge (ADB) provides a mechanism for developers to connect to Android devices from their development environment. Using a set of commands provided by this tool, developers can perform actions such as installing and uninstalling apps, taking device screenshots, accessing app and system logs, file manipulation, etc. I use this tool to install Android apps in the emulator, interact with apps to reproduce failures by sending keyevents, tap, and scroll commands to the emulator, and uninstall the apps. This is needed to build the behavioral taxonomy in my research described in Section 6.2. Further, I use ADB to take app screenshots, record app execution videos and pull them from emulator to my desktop, and collect device information. This is needed to build and run automated oracles described in Section 6.3.3.

### **2.6.2.3 UI Automator**

UIAutomator viewer is a UI inspecting tool that captures the GUI object hierarchy on the screen in XML format. The XML file can be analyzed to find GUI element properties and



state. Such information can be used in UI Automater, Android UI testing framework, to write automated GUI tests [44]. I use UI Automator to access the device and app state information such as GUI element id, class, and element size; and perform touch-based user actions on the device. I also use the app state information in automated test oracles.

## Chapter 3: Related Work

### 3.1 Test Oracle Problem

Bertolino acknowledged test oracle efficiency and effectiveness as one of the challenges in software testing and urged further research on the topic [45]. More recently, Ma’ayan studied 112 Java repositories and documented problems with their unit tests [46]. Tests that mask the existence of faults are unreliable and often useless. A *flaky test* is one type of unreliable test [47–49]. Chapter 4 of this dissertation presents blind tests, another type of unreliable test. Vahabzadeh et al. conducted an empirical study of faults in open source test code [50]. They studied faults that were reported and fixed in the code repository, but did not identify additional faults. They used the term “silent horror test bugs” to describe tests that incorrectly passed (a subset of blind tests). However, since these are the hardest faults to identify (why look at a passing test?), and Vahabzadeh et al. only looked at faults that were found and fixed, they were not able to measure the prevalence of blind tests.

Several tools and metrics have been introduced to help testers develop better test assertions. Xie et al. developed a tool, *Orstra*, to improve automatically generated unit test suites by automatically generating test assertions through regression [51]. Song et al. proposed an eclipse plugin, UnitPlus, which recommends relevant methods to check variable state while writing test assertions [52]. Staats et al. proposed an oracle creation method that ranks variables based on their fault finding capability and monitors those through tests [53]. Loyola et al. proposed a similar approach for automatic oracle creation by selecting a set of variable to be monitored based on the interactions and dependencies observed among the variables [54]. Schuler et al. demonstrated that test oracle quality is an important factor in gauging test quality [55]. They introduced the concept of *checked coverage* as a measure of test oracle quality. Checked coverage is the ratio of statements contributing to the result

checked by the oracle to the statements executed by the test. While the above mentioned metrics and tools focus on solving different aspects of test oracle problem, my empirical study focuses on identifying the cause and prevalence of relatively under-studied problem of blind tests. To what extent the above mentioned tools apply to the problem of blind test is left as a future research direction.

Zhi et al. reported a case study on the inadequacy of test assertions with results that generally agree with result in Chapter 4 [56]. My blind test study differs from their study in several ways: (1) They analyzed tests found in three open source projects, whereas my study is done on tests written by developers. (2) They used mutation operators to generate faults whereas I used both real-life faults that existed naturally in the program and hand-seeded faults. (3) They used mutation location and statement coverage to determine if the fault was found. I used a more precise method—whether the test reported failure on the fault.

Xie and Memon analyzed GUI tests, finding that oracles that check the entire state can detect more faults [57]. They used manually seeded faults whereas we use faults occurred during program development along with hand-seeded faults.

Staats et al. define *oracle soundness* for a program, test case, and oracle as: if an oracle is true for a program with a test case, the specification holds for the program when running the test case [58]. However, this is not always true, as the oracle might observe a subset of the program output or final program state and miss some failures.

Similar to the test adequacy criterion proposed by Koster et al., my study judges the quality of a test based on its *soundness* [59]. A key difference is that Koster et al. looked at whether tests checked all the affected variables or not; whereas this study checks whether a test hits all four steps in the RIPR model to determine if a test is good enough or not. In my study, if a test reaches the fault, creates an error state, and propagates it to output, yet the test assertion cannot reveal it, then the quality of test is considered to be poor.

## 3.2 Related Work to Smart Tests

The literature contains three general techniques to address problems with test suite growth: test suite reduction or minimization, test selection, and test prioritization.

A survey by Yoo and Harman discusses these approaches in detail [60]. Chapter 5 uses test case minimization and selection as intermediate steps, but does not specifically introduce new minimization or selection techniques. It also does not currently address test case prioritization.

Test suite reduction approaches focus on reducing the size of test suites to lower the maintenance cost of large test suite. Chen and Lau proposed minimizing test suites by selecting an essential set of tests that cover requirements that no other tests cover, followed by a greedy algorithm to select more tests [61]. Ammann et al. proposed removing redundant tests until a minimal test set is obtained [62]. The primary focus of my approach is to enable the automated tests to maintain themselves to reduce maintenance costs for developers. The approach also performs test suite reduction whenever a test requirement becomes invalid due to software evolution. Vaysburg et al. performed dependency analysis of Extended Finite State Machines to minimize test suites [63]. In my work, I analyze control flow graphs to detect the impact of software evolution on tests, although my general approach could work just as well with other models and other techniques. A common concern of test suite reduction techniques is to avoid removing a fault-revealing test. To handle this issue, my framework does not permanently remove tests whose requirements are no longer valid; rather, it comments it out and notifies the developer.

Test case selection focuses on selecting a subset of test cases from the test suite after the software under test changes. Various approaches have been explored to perform test case selection using techniques such as data flow analysis [64–66], symbolic execution [67], dynamic slicing [68], CFG graph-walking [69–72], textual difference in source code [73, 74], and modification detection [75]. Chen et al. used a modification-based technique to identify test cases that are affected by modifications to program entities [75]. In my approach,

I compare control flow graphs of two software versions to detect any changes, and select tests that execute the modified statement. I select all test cases that execute the modified statement but only run tests that satisfied a unique set of test requirements.

Test case prioritization is an approach to find an ordering of test cases for execution to get maximum benefit with minimal effort [60,76–79]. I do not address test case prioritization in my framework.

### 3.3 Automated Mobile Testing

This section contains related work on studies of oracles, approaches for building the oracles, and related papers about oracle creation in different domains.

***Studies on Oracles in Mobile Apps:*** Zaeem et al. analyzed 106 bug reports for mobile apps and created a categorization of the oracles needed to detect those bugs [80]. This is the most closely related to my work in Chapter 6. However, my work differs in two key ways: (i) my oracle taxonomy is rooted in analyzing fault patterns *as they manifest through the GUI*—making the taxonomy largely complementary to past work—and (ii) I derive and implement *novel* app-agnostic oracle patterns not identified in their study, by using advancements in machine learning and computer vision techniques. Liu studied cosmetic issues in screenshots of mobile apps and defined oracles for detecting those issues [81]. Compared to Liu’s study, I provide a broader categorization of GUI-oracles, and this categorization can spur the creation of approaches for detecting other types of issues. Packevičius et al. studied text defects in mobile apps and provided an oracle-based approach to detect those defects [82]. Their study provides a narrower class of oracles as compared to my categorization. To the best of my knowledge, my study is the first to provide a broad yet detailed categorization of GUI-based behavior of faults for mobile apps. Additionally, my work shows how the categorization can be used to define test oracles for mobile apps.

***Approaches for Oracles in Mobile Apps:*** A large body of research has focused on defining oracles for detecting issues in mobile apps. Su et al. define an approach for detecting violations of the “independent view property” in Android apps, that is, interacting with

one GUI element should not affect the states of the others [83]. Riganelli et al. defined an approach based on failure patterns to detect data loss bugs in Android apps [84]. Yang et al. proposed a technique to detect cosmetic errors in the layout of Android apps [85]. Eler et al. defined an approach that applied different checks to detect accessibility issues in Android apps [86]. Compared with my work, these and other approaches [87–93] use similar underlying techniques (including optical character recognition, image comparison, image classification, etc.) to define the oracle detection patterns proposed in Chapter 6, but target different oracle types (e.g., energy, accessibility, web, etc.). Further, unlike MAGNETO these approaches are mostly limited to text and image comparison for fault detection. They also rely on bug reports or static analysis to generate expected behavior, whereas MAGNETO relies on the behavioral oracle taxonomy. Finally, other approaches also focused defining oracles that are not GUI-related [94,95].

***Oracles in Other Domains:*** Automated creation of oracles has also been studied in other domains [34]. Some approaches focused on defining GUI-based oracles [96,97] while others focus on different types of oracles [1,23,98,99]. My work is largely orthogonal to these papers and studies in other domains, as my taxonomy specifically focuses on failures that appear in mobile apps.

## Chapter 4: Test Oracle Problem - Blind Tests

An *automated test* is a software component that includes test input values and test evaluation code, known as *test oracles (TO)* [34], usually written as assertions. Automated tests execute against the software under test (*SUT*) and report whether the execution passed (the actual output was the same as the expected output as encapsulated in the TO) or failed (the behavior was incorrect). Automated tests often run in build systems that run daily, hourly, or continuously. Unfortunately, many automated tests are incorrect [34, 100]. Some result in false positives, where the test incorrectly reports the software passed. Others result in false negatives, where the test incorrectly reports the software failed. This chapter reports on research investigating a particular problem with false positives, where incorrect TOs [101] do not notice a failure in the SUT. All of the results in this chapter appeared in my prior paper [22].

A test oracle is *incorrect* if it sometimes reports the wrong result. TOs can be incorrect for many reasons. This chapter focuses on a particular type of TO that is incorrect because it is incomplete. This is a widespread but little studied problem that we call *blind tests*. A *blind test* has an incorrect assertion such that a portion of the output was not observed, and that portion was incorrect causing the result to be incorrect. The incorrect result could be a *false positive*, that is, the software produced at least one output value that was incorrect, but the test assertion did not check that particular output. The incorrect result could also be a *false alarm*, that is, the program behaved correctly on the test but the expected output in the assertion did not include everything that was in the actual output.

For example, consider the Java statements shown in Figure 4.1. The first assertion is correct and will reveal the misspelling in `lastName`. The second does not check the last name, thus is blind to the mistake. The third checks both `firstName` and `lastName` in

the actual output portion of the assertion, but only includes the first name in the expected output, so raises a false alarm even if the value for `lastName` is correct.

```
String firstName = "Aasina";  
String lastName = "Barral"; //faulty spelling  
assertEquals("Aasina Baral", firstName+" "+lastName); //correct  
assertEquals("Aasina", firstName); //blind  
assertEquals("Aasina", firstName+" "+lastName); //false alarm
```

Figure 4.1: Example Java statement and JUnit assertions

Assertions are simple and easy to implement for trivial computations, but are more problematic even for modest procedures. Consider sorting a simple numeric array, for example “`sort([5,3,4])`.” Checking the entire output array (“`assertEquals([3,4,5], result);`”) does not scale to arrays with thousands of elements. The expense is prohibitive, thus the automated test can only check part of the output. Spot checking the first or last element will miss many potential mistakes that may be buried in the middle of the array. Even checking that the result is in correct order will not find failures such as “[3,3,3]” and “[0,0,0]”. Simple assertions are tempting to write and common even for experienced testers, but always have the potential to be at least partially blind. This chapter focuses on blind tests that lead to false positives.

This chapter makes the following noteworthy contributions:

1. We collected 335 automated JUnit tests for two Java programs. These automated tests are written by 58 subjects with varying degree of software testing expertise.
2. We designed and implemented complete test oracles (CTOs) for the 335 automated tests written by the study subjects.
3. We performed several empirical studies of test oracles to quantify, analyze, and categorize blind tests. Our result shows that blind tests are very common problem.
4. We identified and classified some root causes of blind tests.



5. We make all our experimental object programs, including faults, and tests, and CTOs available in the online appendix [102].

We discuss the challenges of creating test oracles in Section 4.1 and motivate the problem. Section 4.2 describes several empirical evaluations of test oracles, followed by observations and results in Section 4.3. Section 4.4 presents threats to validity, and Section 4.5 concludes the chapter.

## 4.1 Challenges of Creating Test Oracles

In Chapter 2, I described test oracle problem and challenges of test oracle. To write correct and complete TOs, programmers and testers need to understand program requirements and correct behavior. It is challenging to write oracles for software that produce complicated outputs, and especially so if the output is non-deterministic. Consider an example SUT that accepts a collection of strings, then returns strings randomly, one at a time, when asked. Given the example partial test below:

```
@Test
public void threeRandomStrings(Calc rs)
{
    rs.add("ICST");
    rs.add("ICSE");
    rs.add("FSE");
    String randomStr = rs.getRandomValue(); //what assertion should be used?
}
```

What should be checked in an assertion? Correct behavior could be any of the three strings, so perhaps the assertion could check whether `randomStr` is equal to any of the three strings. However, that still would not ensure the strings are returned randomly. Checking for randomness requires some complex math—the test method would need to call `getRandomValue()` many times and check whether the distribution of strings returned is

truly random. Such assertions are hard to think of, hard to design, and hard to code.

## 4.2 Empirical Evaluations of Test Oracles

Our first question about blind test oracles is how common are they? Beyond their frequency, we seek to understand what causes tests to be blind. Answers to these questions will help find solutions to the problem. We have carried out several studies of test oracles to try to quantify, analyze, and categorize blind tests. Partners in industry and attendees of Google’s Test Automation Conference [103] have frequently expressed that poor TOs are a major problem in test automation. In a 2010 keynote presentation, Patrick Copeland of Google claimed poor TOs was one of the most significant causes of wasted effort in testing [104]. During previous research with automated tests [1], we found that almost a third of the tests in our study were blind.

For this research, we ask the following research questions about blind tests:

**RQ 4.1** *What percentage of tests are blind?*

**RQ 4.2** *Do some groups of testers write more blind tests than other groups?*

### 4.2.1 Methodology

We collected three separate sets of data on this question, using different subject testers, different object programs, and with different guidance for creating test input values. Details of subject testers and object programs is described in Section 4.2.3, 4.2.4, and 4.2.5. In each, we started with a program module to test (the SUT) and asked engineers to create automated tests. The SUTs had a known collection of software faults.

We introduce the concept of a *complete test oracle* (CTO) to be a test oracle that checks the entire final output space of the SUT. While a CTO may be too large to always be practical, this does give us a ground truth for the test—if the test caused a failure, the CTO is guaranteed to reveal that failure. In general, CTOs may need to include files on disk, databases, internal state variables, etc. In our studies, we narrowed the output space to focus on just the console output of the program and excluded intermediate program state.

We replaced the engineer-written TO of each test with our own CTO. We then compared the result of executing the test with the original TO against the result with the CTO. If the original test did **not** reveal a failure, but the modified test did, then the TO in the original test was blind.

#### 4.2.2 Study 1: Preliminary Analysis

Table 4.1 shows data from a preliminary study that was primarily focused on automating the creation of test input values [1]. The tests were designed and implemented by hand from UML statecharts that described the behavior of the software. Out of a total of 93 automated tests that reached a fault, caused an infection in the program state, and propagated to the output state, only 65 of the TOs **revealed** the failure. That is, 28 tests were successful in that they caused a failure, but their TOs incorrectly reported that the test passed. Put another way, 30.11% of the failure-finding tests were ignored because the TOs were incorrect.

Table 4.1: Study one: Tests that caused failure but did not see the failure

	tests that caused failure	tests that revealed	% tests that revealed	caused failure, did not reveal	% did not reveal
Total	93	65	69.89	28	30.11
UG	30	26	86.67	4	13.33
FG	25	19	76.00	6	24.00
PG	21	12	57.14	9	42.86
FT	17	8	47.06	9	52.94

Not only is this a surprisingly high percentage of faulty TOs, but when broken out by groups, the success rate was surprising. Row UG in Table 4.1 represents undergraduate students, FG represents full-time graduate students with no work experience, PG represents full-time software engineers who are also part-time graduate students, and FT represents

full-time software testers. In our preliminary study, the testers who wrote the most successful TOs are the least experienced (undergraduate students), while the testers who wrote the most faulty TOs are the most experienced—full-time software testers.

These preliminary results, while based on only a small data set, were intriguing enough to convince us to investigate more thoroughly. Thus, we expanded this study with more subjects.

### **4.2.3 Study 2: Students in a Fourth Year Testing Class**

The second study assigned students to write tests for a 100 line method they were already familiar with.

#### **4.2.3.1 Subject and Program Selection**

For this empirical study, we needed subjects who were familiar with the RIPR model and could write JUnit tests. Therefore, we collected tests written by 48 undergraduate students at our university who were taking a senior-level course in software testing. All were either Computer Science or Software Engineering majors and had studied the RIPR model in class. Approximately one-third of the students had some experience as software engineering interns in local companies.

The goal of the study was to compare and assess the quality of test assertions, so we needed all participants to write tests for the same program. The program was based on a quiz scheduling Java application used to schedule retakes of weekly quizzes in their class. Students enter the course number, then pick from a list of retake times available within the following two weeks. The list includes the date, time, location, and quiz id for each available quiz. The list of retake times form part of the input space and is read from two XML files. The students are then prompted to enter their name and the quiz id. The software saves these inputs to schedule a retake. Figure 4.2 shows the example initial screen. The actual scheduler is a web application (shown in Figure A.1 in Appendix A), but students were given a command line version to allow for students who had not programmed Java servlet.

Students wrote tests for a 100 line method within this program that prints the list of available retakes. I provide the method used in the study in listing A.1 in Appendix A.

```
*****  
University quiz retake scheduler for class Software Testing  
*****
```

```
You can sign up for quiz retakes within the next two weeks.  
Enter your name (as it appears on the class roster),  
then select which date, time, and quiz you wish to retake  
from the following list.
```

```
Today is THURSDAY, FEBRUARY 28.  
Currently scheduling quizzes for the next two weeks,  
until THURSDAY, MARCH 14  
RETAKE: THURSDAY, FEBRUARY 28, at 10:00 in Mason Hall  
    1) Quiz 4 from TUESDAY, FEBRUARY 19  
    2) Quiz 5 from TUESDAY, FEBRUARY 26  
RETAKE: TUESDAY, MARCH 5, at 15:00 in EB 5321  
    3) Quiz 4 from TUESDAY, FEBRUARY 19  
    4) Quiz 5 from TUESDAY, FEBRUARY 26  
    5) Quiz 6 from TUESDAY, MARCH 5  
RETAKE: WEDNESDAY, MARCH 6, at 15:30 in EB 4430  
    6) Quiz 5 from TUESDAY, FEBRUARY 26  
    7) Quiz 6 from TUESDAY, MARCH 5  
RETAKE: THURSDAY, MARCH 7, at 10:00 in Mason Hall  
    8) Quiz 5 from TUESDAY, FEBRUARY 26  
    9) Quiz 6 from TUESDAY, MARCH 5
```

Figure 4.2: Quiz retake scheduler screen

#### 4.2.3.2 Program Requirements

The students used this program for the class to schedule quiz retakes, so were familiar with its behavior. The course features weekly quizzes, and students are allowed to take an alternate version of a quiz within two weeks for 80% of the possible grade. The version we

provided had known faults. We describe key requirements of the program here.

- Requirement 1: Students can only retake a quiz within 14 days from the date of the first attempt, counted from the date and time of the in-class quiz. The two week retake period can be extended if there is a break week during that period.
- Requirement 2: The application shows retake opportunities available within the next two weeks from the current date and time. The application also prints a message indicating the last day of the period (for example, “*Today is THURSDAY, FEBRUARY 28. Currently scheduling quizzes for the next two weeks, until THURSDAY, MARCH 14*”).
- Requirement 3: A *skip week* is a week with no classes, such as spring break. There are no retake opportunities or quizzes during skip weeks.
- Requirement 4: If there is a skip week within the next two weeks, the software extends the “currently scheduling” period by an additional week.
- Requirement 5: If there is a skip week within the next two weeks, the software extends the retake period to three weeks. For example, for a quiz originally given on February 19, if February 20 through February 27 is a skip week, the last retake date is extended from March 5 to March 12.
- Requirement 6: If a skip week is coming up, the application informs the student there will be no retakes or quizzes during that week. This message is shown between the last retake option before the break and the first retake option after the break.

#### **4.2.3.3 Known Faults**

The program had five known faults, all of which occurred naturally when the program was developed. The subjects had access to the program requirements, but were not told the software had faults. The faults are as follows:

1. The faulty method only checked if the last day of the two week scheduling period is in the skipped week, but does not check if the current day is in the skipped week or if there is a skip week between the current day and the last day of the period. This fault violates requirements 1, 2, and 4.

2. The last retake day for a quiz is not extended when there is a skip week between the in-class quiz and the last retake day for that quiz. This fault violates requirement 1 and requirement 5.
3. A retake is available on the 14th day after the in-class quiz, but **after** the quiz time. That is, the retake is available for 14 days plus several hours. This does not conform to requirement 1.
4. The message about skipped week is printed between the wrong retake options. This violates requirement 6.
5. The application prints retake opportunities during the skip week. This violates requirement 3.

#### 4.2.3.4 Data Collection

The Java program was given to the students and they submitted their JUnit tests electronically. The students wrote their tests in teams of three or four students; we consider each team to be one subject. We collected 137 automated tests from 23 subjects. The subjects were unaware of the study being conducted and the collected data were de-identified before analysis to ensure anonymity. The quality of the tests, including the test oracles, was **not** measured as part of their grades.

#### 4.2.3.5 Test Evaluation

To ensure we could associate each failure with its triggering fault, each fault was embedded into a separate version of the program. Thus we had six versions of the program. We also created complete test oracles (CTOs) that checked the entire final output state of the program excluding the intermediate state. We used the CTOs to create second versions of each test. The CTO versions of the tests have the property that if the test resulted in a failure they are guaranteed to reveal the failure, even if the original test's oracle did not.

We ran each test, and its alternate, on each faulty version and the original program ( $137 \times 2 \times 6 = 1644$  executions). We instrumented the program to track whether each test

reached, infected, propagated, and revealed the failure. Comparing results of an original test with its CTO version let us identify tests that caused failure but did not reveal the fault.

#### 4.2.4 Study 3: More Undergraduate Students

Our third study used a different group of undergraduate students who took the same senior-level testing course in a different semester. The subjects were similar and the procedure was the same as in study 2, however this study used a different object program.

The object was a small calendar program that prints a month in calendar format given two integers representing a month and a year. The calendar program source code with faults is in listing A.2 in Appendix A. The output created from the inputs “10 2019” is shown in Figure 4.3. The first line of the output displays the month and the year entered e.g., October 2019. It prints abbreviations for seven days of the week as “S M Tu W Th F S” in the next line. Then it prints the dates in the month from 1 to 28, 30, 31, or 32, depending on the month.

```
October 2019
S M Tu W Th F S
      1  2  3  4  5
6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31
```

Figure 4.3: Calendar output

We hand-seeded five faults, as described below. The program requirements and the implementation are much simpler than the quiz retake scheduler. For example, calendar application only takes command line input, where as the quiz retake scheduler reads command line input and xml files as input. As such, we expected students to infect, propagate,



and reveal the faults with greater frequency.

#### 4.2.4.1 Seeded Faults

We seeded the following five faults into the program.

1. Fault 1: The program had no input validation, and threw run-time exceptions if non-integer values were entered, or if a month value less than 1 or greater than 12 was entered. Negative values for the year return reasonable results, although the program does not switch to a Julian calendar if a year before 1582 is entered. (This fault was naturally occurring.)
2. Fault 2: In the leap year calculation, the predicate  

```
"((year%4 == 0) && (year%100 != 0))"
```

was changed to  

```
"((year%4 == 0) || (year%100 != 0))"
```
3. Fault 3: We changed the number of days in August from 31 to 30.
4. Fault 4: In the computation for which day of the week a particular date falls on, we changed the number of months from 12 to 10.
5. Fault 5: We changed the number of days in February in leap years from 29 to 30.

#### 4.2.4.2 Data Collection and Test Evaluation

This study used the same process as in study 2 in Section 4.2.3. We collected 184 tests from 30 subjects and embedded each fault into a separate program. This required  $184 \times 2 \times 6 = 2208$  executions.

#### 4.2.5 Study 4: Professionals at a Software Engineering Company

Our third study used engineers at a local software company. The company asked that we keep its name private. They included a mix of developers and full time testers, who volunteered their time without compensation. Most were early-career professionals 5 to 7 years out of college. All had college degrees in computer science or software engineering.

We used the same Java program as in study 2 and shared the same requirements with the testers. The professionals wrote the JUnit tests in small teams of 1 to 5. We collected 14 automated tests from 5 subjects. Unlike the previous students study, the subjects were aware a study was being conducted. The data were anonymized before being analyzed. The rest of the study was the same as study two.

Table 4.2: Study two (students): Number of tests that reached each condition in the RIPR model for the five faults—137 total tests

	Fault 1	Fault 2	Fault 3	Fault 4	Fault 5	Total
Reachability	83	78	78	80	78	397
Infection	5	4	4	3	11	27
Propagation	5	4	4	3	11	27
Reveal	2	2	0	1	3	8

### 4.3 Observations and Results

The quality of the test oracles from our student subjects in study 2, as shown in tables 4.2 and 4.3, was even lower than in the preliminary study. Out of the 137 tests, an average of 79.4 reached the faulty location, between 3 and 11 tests caused the fault to infect the program state and propagate to output, and fewer than 4 tests revealed the faulty behavior. Fault #3 was not revealed by any tests written by the subjects but was revealed by the complete test oracle. We think this is because not all subjects understood requirement 1 completely. Requirement 1 specifies that the 14 days period considers the time of the in-class quiz as well. It’s also interesting to note that all infections propagated to output, that is, no faults were masked.

Table 4.3 breaks out the percentage of tests that caused a failure (propagated an infected state) and that also revealed the failure. At the high end was fault #2, for which failures were revealed half the time, and at the low end was fault #3, for which none of the four

failures were revealed.

Table 4.3: Study two (students): Frequency of propagating failures that were revealed

	Propagated	Revealed	% of tests that revealed after propagation	% of tests that did <b>not</b> reveal after propagation
Fault 1	5	2	40.00%	60.00%
Fault 2	4	2	50.00%	50.00%
Fault 3	4	0	0.00%	100.00%
Fault 4	3	1	33.33%	66.67%
Fault 5	11	3	27.27%	72.73%
Average	5.4	1.6	29.62%	70.38%

In testing, a test is considered to be successful if it causes the software to fail. Yet, overall in this study, 70% of the “successful” tests did not reveal the failures that they found—that is, they were blind.

Study 3 used a simpler program with more straightforward behavior. It was based on an assignment from an introductory programming course. Unlike the quiz retake scheduler, calendar only reads two integer inputs, and does not read or write to an external file. The logic is also less complicated and the faults were less “subtle,” that is, a higher percentage of input values would result in failure.

Thus it is not surprising that more tests caused a failure, and that a higher percentage of tests revealed the failures (61%).

The results from professional software engineers were less encouraging. Table 4.6 shows that a higher percentage of tests reached the faults (86%), and a higher percentage created an infected program state and propagated to incorrect output (20%). Yet only one test for one fault revealed the failure that it caused! As with Table 4.3, Table 4.7 shows the percentage of failure-causing tests that revealed. In this study 95% of the tests that caused failure were blind. As in the preliminary study, the students in studies 2 and 3 created better test oracles than the professionals in study 4.

Taken together, these studies are clear and convincing that blind tests are a major problem for test automation. Many tests are wasted because their test oracles are incorrect. Not only does this waste valuable resources, but we lose the ability to improve our software by correcting the faults. Thus, our software is less reliable.

Table 4.4: Study three (students): Number of tests that reached each condition in the RIPR model for the five faults—184 total tests

	Fault 1	Fault 2	Fault 3	Fault 4	Fault 5	Total
Reachability	175	113	113	113	113	627
Infection	63	28	6	13	54	164
Propagation	63	27	6	13	54	163
Reveal	63	18	3	5	27	116

Table 4.5: Study three (students): Frequency of propagating failures that were revealed

	Propagated	Revealed	% of tests that revealed after propagation	% of tests that did <b>not</b> reveal after propagation
Fault 1	63	63	100.00%	0.00%
Fault 2	27	18	66.67%	33.33%
Fault 3	6	3	50.00%	50.00%
Fault 4	13	5	38.46%	61.54%
Fault 5	54	27	50.00%	50.00%
Average	32.6	23.2	61.03%	38.97%

We cannot be certain why students’ tests consistently performed better than professionals’ tests. Our working theory is that test automation has only recently been widely taught at universities (often in early programming courses). Thus current students learned test automation with both theory and practice, while many professionals learned JUnit syntax on the job without deep study. This is only speculation, however, and we hope that further work can shed more light on this question.

Table 4.6: Study four (professionals): Number of tests that reached each condition in the RIPR model for the five faults—14 total tests

	Fault 1	Fault 2	Fault 3	Fault 4	Fault 5	Total
Reachability	14	12	11	11	12	60
Infection	4	2	2	2	4	14
Propagation	4	2	2	2	4	14
Reveal	0	0	0	0	1	1

Table 4.7: Study four (professionals): Frequency of propagating failures that were revealed

	Propagated	Revealed	% of tests that revealed after propagation	% of tests that did <b>not</b> reveal after propagation
Fault 1	4	0	0%	100%
Fault 2	2	0	0%	100%
Fault 3	2	0	0%	100%
Fault 4	2	0	0%	100%
Fault 5	4	1	25%	75%
Average	2.8	0.2	5%	95%

### 4.3.1 Root Causes of Blind Tests

After identifying blind tests, we analyzed the root cause for why each test missed its failure. We then grouped them into four categories. We describe them, with examples, below.

1. *Test and source code reuse*: We found one cause of blind tests to be reuse of test code.

Some testers reused test assertions from a previous test, but without appropriately changing the assertion for the new test. Other testers reused the implemented source code when putting expected outputs into the test oracle. If testers reuse assertions without carefully analyzing the new test inputs, it is easy to create blind tests.

For example, one fault in the Calendar application incorrectly assigned 30 days to August instead of 31.

```
int[] days = { 0, 31, 28, 31, 30, 31, 30, 31, 30, 30, 31, 30, 31 };
```

Some testers also used 30 instead of 31 while creating tests, essentially copying the expected output from the actual output. Thus they had the expected output as:

```

expectedOutput = " S  M Tu  W Th  F  S \n" +
                  "           1  2  3 \n" +
                  " 4  5  6  7  8  9 10 \n" +
                  "11 12 13 14 15 16 17 \n" +
                  "18 19 20 21 22 23 24 \n" +
                  "25 26 27 28 29 30 \n";

```

2. *Misunderstood program requirements:* Some tests were blind because the tester misunderstood what the correct program behavior should be. Although these test oracles may have observed the correct part of the output space, the assertions were written with incorrect behaviors.

For example, requirement 4 in study 2 states that if there is a skip week (a week with no classes, such as spring break) within the next two weeks, the software should display an extended retake period. A tester coded the following into a test:

- *startSkip* = "2019-3-1";
- *endSkip* = "2019-3-7";
- *expectedOutput* = "Today is THURSDAY, FEBRUARY 28. Currently scheduling quizzes for the next two weeks, until THURSDAY, MARCH 14."
- *actualOutput* = "Today is THURSDAY, FEBRUARY 28. Currently scheduling quizzes for the next two weeks, until THURSDAY, MARCH 14";
- `assertEquals(expectedOutput, actualOutput);`

Since the program's output matched the expected output, the test was considered to have passed. Unfortunately, the tester misunderstood the point of the *skip week*. The correct output should have been:

- *correctOutput* = "Today is THURSDAY, FEBRUARY 28. Currently scheduling quizzes for the next two weeks, until THURSDAY, MARCH 21";

As a result, the test was incorrectly marked as passing.

3. *Technical inexperience:* We also observed that testers sometimes were not familiar

enough with the programming language or the testing tool. Programmers who had difficulty understanding the code and did not understand how the test assertions worked had difficulty setting up test inputs and writing test assertions. This led to the creation of test oracles that either did not observe the output space correctly or observed an unimportant section of the output space. This problem was **not** unique to students; but also happened with professional developers and testers.

The subject programs used in studies 2, 3, and 4 print to standard output (using the Java `out.println` method). Printed output can be captured using the `System Rules` API or the `ByteArrayOutputStream` class, but some testers did not know how, did not capture outputs correctly, and thus did not see failures. For example, one tester rewrote the `Calendar` program so that instead of printing the output, it returned a string as an output. Another student subject simply wrote no assertions at all, so missed **all failures**.

4. *Lack of testing knowledge*: We found that some subjects were confused about how to design a test oracle, and thus created blind tests. This was more common with professionals than students. For example, some tried to validate the implemented code instead of verifying its correctness through testing. In fact, they assumed the implemented code was correct and the goal of testing is to make sure that it works without crashing.

For example: Fault #5 in `Calendar` was that it assigned the number of days in February to be 30 instead of 29 in leap years.

```
if (month == 2 && isLeapYear(year))
    days[month] = 30;
```

Some of our testers copied the assertions from the output of the incorrect program:

```
expectedOutput = " S  M Tu  W Th  F  S\n" +
                  "           1  2 \n" +
                  " 3  4  5  6  7  8  9 \n" +
                  "10 11 12 13 14 15 16 \n" +
                  "17 18 19 20 21 22 23 \n" +
                  "24 25 26 27 28 29 30 \n";
```

## 4.4 Threats to Validity

This section summarizes threats to validity and what we did to mitigate them. Some subjects might not have known, or not fully understood the RIPR model, leading to incomplete tests. Since this study relies on the model to assess test quality, we mitigated this threat by recruiting subjects who were either familiar with the concept of RIPR model or had 5 to 7 years of professional software developer experience. On average, just 71% of the total test cases reached each fault. The remaining 29% did not reach a fault, thus could not be analyzed for test oracle quality.

The overall number of subjects and test cases is another threat to validity. The study required quite a bit of hand analysis, including creating the complete test oracles and determining why some test oracles did not succeed. This limited the total number of subjects and test cases that could be used. A follow-up study that took a simpler approach would be less precise, but might be able to analyze more tests. Another potential threat is the programming language. We used Java, and it is possible that the results might be different with other programming languages.

Although we gave subjects detailed specifications of the application under test, it is possible that some did not understand the expected behavior.

Finally, we observed that some of the students did not put out as much effort as might be hoped. This probably affected the quality of the test inputs more than the quality of the test oracles, the true target of the study.

## 4.5 Conclusions and Future Work

The most important conclusion about this study is that blind tests are very common. This leads to waste and inefficiency, and contributes to the billions of dollars lost every year on faulty software [8]. Although not as widely studied as flaky tests [20], blind tests may be a



more common problem.

The list in Section 4.3.1 is a strong starting point, but as yet incomplete. Future work should aim to identify and classify additional root causes of blind tests.

As we continue to grow our understanding of blind tests, there are three future directions to address the problem. First, educate software engineers, including developers and testers, as to how to properly write effective test oracles. Developing educational materials and evaluating their value on students as well as professional software engineers can be beneficial.

Next, develop techniques to automatically detect TO problems. This can be done with both static and dynamic analysis. Static analysis techniques such as slicing can be used to identify parts of the output domain that can be modified by specific test inputs. If those output values are not checked, the TO could be blind. CTOs can be generated dynamically and used sporadically. CTOs are normally too expensive to use every time a test is run, but they could be used on a probabilistic basis to compare with the existing TO. If the outputs differ, the TO probably has blind spots.

Finally, adapt automatic program repair (APR) techniques to test oracles [105, 106]. This context is smaller than general APR and much of what a test oracle should check can be determined through analysis techniques such as slicing and data flow.

## Chapter 5: Test Maintenance Strategy - Smart Tests

This chapter presents a novel, holistic solution to the problem of test suite management. All of the results in this chapter appeared in my prior paper [23]. As software evolves, its associated automated tests must also evolve. For each change, existing test fall into one of four categories:

1. test needs to rerun to verify the change,
2. test does not need to be rerun,
3. test needs to be changed to adapt to new syntax or behavior, or
4. test is no longer needed and can be deleted.

We collectively call these activities *test suite management*. These topics have been previously studied and useful algorithms and techniques have been proposed [66,67,75,107,108], yet much of the research is piecemeal and few ideas have been adopted in practice. Crucially, all prior work depended on human testers to do much of the work; part of the novelty of this research is to move the burden of making decisions and taking action from the human to individual tests.

The result is that software developers currently manage their test suites by hand, with all the expected difficulties. Sometimes all tests are rerun for every change. This is fine for small programs with small test suites, but does not scale to programs with millions of lines of code and thousands or tens of thousands of tests. Sometimes the decision of which tests to rerun is made intuitively, with the expected loss of fidelity and effectiveness when some tests that need to rerun are skipped and others that should not be rerun are not skipped. Often tests that need to be changed to reflect changes in the software are either ignored, if they still compile and execute to completion, or simply deleted if they do not. This leads to tests that are no longer useful or the loss of valuable tests. And finally, when test

management is done by hand, few tests are deleted when no longer needed—they simply stay around being rerun for no good reason, sometimes for years.

The term *test bloat* is used for test suites that continue to grow and contain increasing numbers of useless and unnecessary tests. Test bloat wastes valuable resources. Computer resources are wasted when useless tests are run. Worse, human resources are wasted when tests fail because of the test, not the software, and when humans spend increasing amounts of time poring through voluminous test results that are full of noise, looking for the ever-shrinking signal. When coupled with flaky tests [20,47] and blind tests [22], these problems lead to systemic and industry-wide waste, in turn making software more expensive and less reliable.

This chapter presents a novel self-management approach to maintaining test suites that is inspired by two conference presentations [109,110]. We propose a practical and comprehensive strategy based on having individual tests self-manage. We say that a test is *self-aware* if it has access to its purpose in an actionable way. For example, if its purpose is to cover a specific edge in a control flow graph (its test requirement), then the test must encode the edge or edges that it covers and be able to access information as to whether that edge is still present after the software changes. Or, if the test’s purpose is to verify a specific functional or nonfunctional requirement, it needs to encode the requirement it verifies and be able to access information about whether the requirement is affected by the software’s change. This information is typically stored in “soft” form, such as in natural language documentation such as comments, or notes on paper, or even lost completely. For example, software engineers often create tests for a specific purpose but then do not document that purpose.

Further, we say that a test has *self-determination* if it has the ability to decide what should happen after the software changes. That is, depending on its purpose and the change, should the test rerun, not run, be changed, or be deleted? At present time, we expect that a test that needs to be changed to reflect the software’s change must be changed by a human. In the future, we plan to investigate the possibility of using automatic program

repair techniques to “repair” tests that no longer compile or run.

This chapter makes following contributions:

1. We designed and implemented a practical and automated approach for discovering and tracking statements covered by automated tests.
2. We implemented an automated approach to create and parse control flow graphs from the source code and identify the graph coverage requirement satisfied by the automated tests.
3. We designed and implemented a framework that uses the coverage information of individual unit tests to decide appropriate action as the software evolves.
4. We evaluated the feasibility of our framework by analyzing its run time and accuracy of the action recommended.
5. We make our framework and experimental data available in our online appendix [111].

We discuss background and challenges of test suite management in Section 5.1. Section 5.2 describes the proposed framework for test self management. Our empirical study is discussed in Section 5.3, followed by results in Section 5.4. Section 5.5 presents threats to validity, and Section 5.6 concludes the chapter.

## 5.1 Background and Challenges of Test Suite Management

Every program has a suite of tests that are designed to verify that the program’s behavior is as expected. As the program evolves, some tests are added, some are modified to reflect the program’s changes, some are no longer needed and deleted, and others are unchanged. Running all the tests in a test suite after every program change costs time and effort, and is prohibitively expensive for software at scale. Thus, testers try to only run tests that may behave differently on the modified program.

### 5.1.1 Managing Tests by Hand

Automated tests encapsulate test inputs and expected results, allowing them to run automatically and report results. However managing tests as software evolves is almost entirely

done by hand. Each test was designed for some purpose, but that purpose is seldom recorded in a machine-readable form, putting the responsibility of investigating, evaluating, understanding, and remembering the purpose of tests on the developers. Although researchers have published algorithms for evolving and managing test suites [107, 108], the algorithms are not available in useful tools that practical testers can integrate into their typical workflow. Thus, they are left with the burdens of investigating the cause of a failing test and deciding which tests need to be rerun after the software changes.

For every software change, the developer has to:

1. understand the program's logic and flow
2. understand why each test was created
3. evaluate all tests to see if the change affects the test's purpose
4. decide which tests to rerun, modify, and discard

Performing these tasks by hand is slow, error-prone, and relies on tester expertise. Despite numerous research advances, the field has not successfully deployed tools for developers. As part of this dissertation research, I designed, developed, and evaluated novel infrastructure that can allow tests to perform the above four tasks and significantly reduce developers' burden.

### 5.1.2 Information Needed

A *smart* test can decide if it needs to be run, ignored, modified, or discarded after the software is changed. To do so, it needs to answer five questions:

1. What are the test requirements of the program?
2. What does the test currently cover?
3. Does the test's current coverage match the test requirement?
4. What has changed in the program?
5. Do the program changes affect the test's purpose? If so, how?

This information must be available in a form that can be processed by software.

### 5.1.3 Central Management vs. Test Responsibility

Automated tests are widely used in the software industry, and usually managed by hand. Because modifying and removing out-of-date tests is time-consuming and complicated, test suites are often not really managed, they merely grow. This leads to *test suite bloat*, where the test suite grows unchecked and becomes increasingly harder to manage. Bloated test suites include tests that fail because they no longer match the software, which in turn leads to testers not noticing tests that truly fail.

Researchers have proposed techniques to centrally manage test suites. These include algorithms to prioritize tests for ordered execution, algorithms to remove tests that are no longer needed, and procedures to identify code elements that are not currently covered [76–78, 107, 112–115]. However, central management is expensive and cumbersome, and tools that are useful in practice are not available.

### 5.1.4 Test Self-management

To move from managing test suites centrally by hand to test self-management, we need tests to be self-aware and to have self-determination. A self-aware test must know its traceability information such as what requirements or code elements it covers (its *purpose*), and be able to discover what has changed in requirements or program. This information is typically kept in inaccessible documentation, not kept at all, or at best, re-computed when needed. Self-determination means that the test can check the available information and decide, after the software changes, whether the test needs to run as is, not run, be changed, or be deleted. We call tests with self-awareness and self-determination *smart*, as they have the ability to manage themselves.

## 5.2 A Framework for Test Self-management

This section describes a test self-management framework. The framework consists of five automated, sequential, steps to answer the five questions listed in 5.1.2. We first give a

high-level description in Section 5.2.1, followed by a more algorithmic-level description in Section 5.2.2.

### 5.2.1 Test Framework’s Five Step Process

Figure 5.1 illustrates the five steps, and we discuss each below.

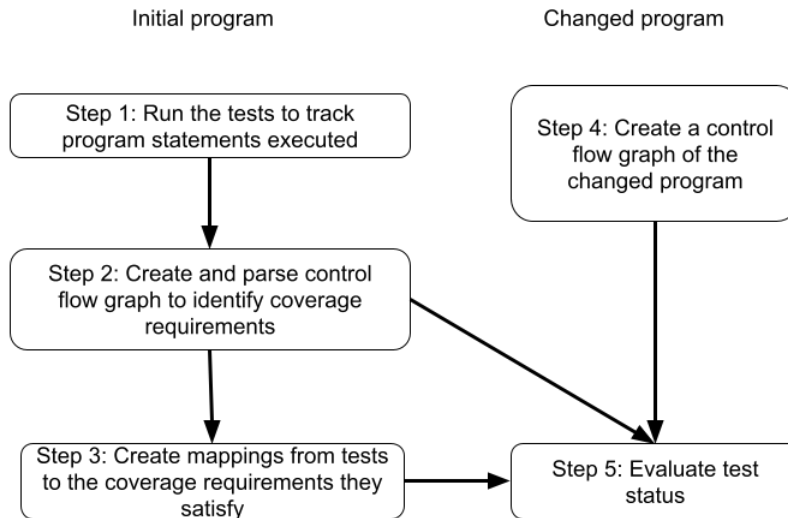


Figure 5.1: Test self-management framework

Step 1) *Run the tests to track program statements executed:* This lets us track which statements were covered by each test, allowing us to identify each tests’ purpose and evaluate its status, as discussed in steps 3 and 5.

To track statement coverage, we modify the `.class` file using bytecode injection at runtime to inject tracing code immediately before the class is loaded and run. We use the bytecode manipulation framework ASM [116], which provides common bytecode transformation and

analysis algorithms to build code analysis tools.

Step 2) *Create and parse the control flow graph to identify coverage requirements:* We compare the control flow graph (CFG) of a program before and after program evolution to detect precisely how the program was changed. This comparison lets automated tests be aware of program evolution.

We use a cross-platform tool, Progex [117], to create control flow graphs. Progex reads program source code, then generates the control flow graph and exports it into a file format for graphs such as DOT [118], GML, and JSON [119] (we use DOT). Following is an example of a Progex-generated CFG for SourceClass.java in DOT file format:

```
digraph SourceClassCFG {
// graph-vertices
v1 [label="17: SourceClass(int qtyOnHand)"];
v2 [label="18: this.qtyOnHand = qtyOnHand"];

// graph-edges
v1 -> v2;
}
```

Each node in the CFG, v1 and v2 in the example, represents a single line of code. Each node has the line number and the Java statement it represents as its label. Connections between nodes are under **graph-edges**. In this example, the graph has an edge from node v1 to node v2.

Step 3) *Create mappings from tests to the coverage requirements they satisfy:* Coverage criteria provide test requirements, and in turn, each test has a specific purpose—to satisfy one or more test requirements.

Although the concepts behind self-determining tests can be applied to any type of test requirement, whether functional or non-functional, our research focused on one test coverage criterion, edge coverage (ECC). Many available tools measure ECC and it is commonly used



in industry. The test requirements for ECC are the edges in a graph, that is, a pair of Java statements.

We use an automated script to identify the ECC requirements. The script uses the control flow graph generated by Progex [117] to create a list of edges that must be traversed. Based on the statements a test executed (from step 1), we map the test to the requirements (pairs of Java statements) it satisfied to document the tests' purpose. This mapping is used as the program evolves.

We use another automated script to create the mapping between a test and the requirements it satisfies. The script uses the test requirements identified for a criterion and the test coverage information from step 1 and produces a mapping in JSON format. JSON is a standard, lightweight, data format that uses human readable text to store data in a map structure [119]. Following is a sample JSON mapping.

```
[
  { "info": "27-04-2020_17_01_08"
  },
  { "requirements":
    { "TR1": "17,18" }
  },
  { "testCoverage": {
    "testAdd": ["TR1"]
  }
}
]
```

The **info** line gives the creation date, and **requirements** contain the test requirements, in this case, to cover edge (17, 18). Thus, this JSON script indicates that the test method *testAdd()* satisfies test requirement TR1.

Step 4) *Create a control flow graph of the changed program:*

We compare CFG of the older program version with that of the newer program version to identify if there are any changes in the program. If any change is detected, it can then

be analyzed to see if it affects the test requirements, and in turn, the purpose of the tests, such as cover statements 1-10 in method `foo_bar()`.

Step 5) *Evaluate test status*: For a test to decide what to do after a change, the test needs to know how the program's source code was changed. In our system, this decision is based on the difference between the old and new CFGs. If the change is more than an addition of white space we say the change is *substantial*, and the effects of the change on tests need to be evaluated. Once the test knows its purpose and is aware of the change, it can then compare the two and decide if the change affects its purpose. The test evaluates the change with respect to its purpose to decide. If the test's purpose is affected, it evaluates the extent of the change and acts accordingly.

### 5.2.2 Details of the Test Framework Process

The previous steps were described at a high level, in terms of goals and results. Now we discuss how these steps are carried out. We used Python and bash scripts to implement the framework.

**1) Detecting software changes:** We compare the control flow graph of the modified program with the prior CFG to detect changes in the code. This includes comparing every statement in the old and new versions in case the change affected the line number or node in the CFG. For efficiency, we identify individual program methods and compare statements within a method. We use the `SequenceMatcher` class [120] of the `diffLibPython` library [121] to make this comparison.

`SequenceMatcher` compares pairs of sequences of any data type. The `ratio` method of `SequenceMatcher` returns a measure of two sequences' similarity as a float in the range [0, 1], where 1 represents an exact match and 0 represents no match. This ratio is calculated as:  $ratio = 2.0 * \frac{M}{T}$ , where  $M$  is the number of elements that match and  $T$  is the total number of elements in both sequences.

We provide two statements, representing a node in the two control flow graphs, as input to the `ratio` method. The sequence is simply the characters in the statement, for

example, the statement “`x = a*b;`” is a sequence of 8 characters. A higher ratio means more similarity. Low similarity could mean one of two things. First, two different unmodified statements could be being compared, for example, the statement on line 1 could be being compared with line 10. Second, a statement is compared with its modified version in the modified program. The SequenceMatcher documentation [121] says to interpret a value over 0.6 to mean that the sequences are a close match. We follow this recommendation to make the following inferences:

- i. If the ratio between two Java statements is 1, they are identical and no change has been made. We categorize these as *perfect match statements*.
- ii. If the ratio between two Java statements is 0.6 or above and below 1, we categorize them as *close match statements*.
- iii. If the ratio is below 0.6, they have low similarity and need to be evaluated further.

**2) Analysis of the program evolution:** We consider 4 types of program evolution: (1) no substantial change from the previous program version, (2) statements in the previous program were modified, (3) statements in the previous program were removed, and (4) statements were added in the new version. The previous step (detecting software changes) creates two categories of statements: perfect match statements and close match statements. These categories need to be further analyzed to understand what kind of program change they represent:

- i. *Was there a substantial change in the code?*

If all the statements from the previous program are present in the changed program and were categorized as perfect match statements, no change was made to the program.

- ii. *Were statements in the initial program modified?*

If statements from the previous program were categorized as close match, the statements were modified. This means the program was modified.

- iii. *Were statements removed from the initial program?*

If statements from original program were not categorized as either perfect match or close match, either the statement from the previous program was significantly changed or it

was removed.

iv. *Were statements added to the modified program?*

If statements from the modified program were neither a close match nor a perfect match, they were added to the modified program.

**3) Impact analysis of the program change on test:** We categorize program changes into four possible types.

i. *No changes to the previous program:*

No test requirements were affected and no tests need to be run or modified.

ii. *Some statements were modified in the previous program:*

We assume edge coverage is being used, so a change to a statement implies a change to all edges that statement appears in. Thus, all tests that cover those edges need to be rerun, and some may need to be modified. Currently, our framework does not decide if the test needs modification, instead it notifies the developer with the code differences highlighted and lets the developer decide the appropriate action.

iii. *Statements removed from the previous program:*

If a statement is removed from the previous program, then any edge it appeared in are no longer present in the CFG, and those test requirements are gone. Tests that covered that edge may no longer be needed. If the change removes only a single test requirement (one edge), then a test that satisfies that test requirement needs to be rerun and possibly modified. If a change removes several test requirements, for example, deleting a complete method, then tests that satisfy those test requirements are no longer needed and can be discarded.

iv. *Statements added to the previous program:*

If statements were added, one or more new test requirements were created and existing test requirements might be affected. This means all the tests that satisfy the affected test requirements need to be rerun, and new tests may be needed.

We automate these three steps in a Python script. The script takes the control flow graph of the original and modified program, and uses SequenceMatcher to identify the statement

level changes. Then our script uses the mapping between tests and test requirements from step 3 and the similarity ratio from SequenceMatcher to let the tests to decide what action to take. If the test needs to be changed, it informs the (human) developer, including a message such as:

```
code in original program:  if (qty < 0)
modified version:  if (qty > 0)
```

If multiple tests need to be rerun, we compare requirements satisfied by the affected tests and if there is an exact match of satisfied requirements, we run only one of the affected tests.

This is a relatively simple approach that could be replaced or augmented by one of the more sophisticated approaches discussed in Chapter 3.

We use Progex [117] to create a control flow graph of the modified version of the program. We then use the original and modified program CFGs to evaluate the test status, as described in step 5.

Figure 5.2 shows a sample result reported by the test management framework. The result contains information in natural language and it is displayed to the developer through a command line interface. The result lists all the tests that were run by the framework. This is followed by the “SOURCE CODE STATUS” section. This section lists the changes detected in the source code. If no change is detected, it says “No change detected in the source code.” Next is the “TEST CODE STATUS” section. This section first lists all the requirements affected by the change in source code as “ $TR_n$ ”. TR stands for test requirement. The mapping of a TR to program statements is in the JSON file generated in step 3. Next the report has an “Affected Tests” section, which gives a recommended action, affected test name, and the reason for the recommended action. Consider the following example:

```
Rerun test:-----> testAddition to check requirement {'TR3','TR2'}
```

```

RUNNING TEST testString
RUNNING TEST testQuadratic
RUNNING TEST testSerial
RUNNING TEST testQuintic
=====
SOURCE CODE STATUS
=====
Line '70' deleted in new version of PolynomialFunction
=====
TEST CODE STATUS
=====
=====
Affected Requirements
=====
TR2
TR3
=====
Affected Tests
=====
Rerun test:-----> testAddition to check requirement {'TR3', 'TR2'}
Rerun test:-----> testConstants to check requirement {'TR3', 'TR2'}
Rerun test:-----> testLinear to check requirement {'TR3', 'TR2'}
Rerun test:-----> testMath341 to check requirement {'TR3', 'TR2'}
Rerun test:-----> testMultiplication to check requirement {'TR3', 'TR2'}
Rerun test:-----> testQuadratic to check requirement {'TR3', 'TR2'}
Rerun test:-----> testQuintic to check requirement {'TR3', 'TR2'}
Rerun test:-----> testSerial to check requirement {'TR3', 'TR2'}
Rerun test:-----> testString to check requirement {'TR3', 'TR2'}
Rerun test:-----> testSubtraction to check requirement {'TR3', 'TR2'}
Rerun test:-----> testfirstDerivativeComparison to check requirement {'TR3', 'TR2'}
=====
Minimal Tests Re-ran
=====
Reran test:-----> testString
Reran test:-----> testQuadratic
Reran test:-----> testSerial
Reran test:-----> testQuintic

```

Figure 5.2: Test self-management result

Here, the recommended action is `rerun` for the affected test `testAddition`. The reason for the rerun recommendation is to check requirement `{‘TR3’,‘TR2’}`. Next, the report lists the name of minimal test that were re-run. Minimal test to be re-run is decided by comparing the requirements satisfied by the affected tests. If there is an exact match of a satisfied requirement, we run only one of the affected tests.

### 5.2.3 Scope of the Framework

We developed our test framework for Java projects built in Maven [122] and unit tests written in JUnit. We used the ASM [116] framework to perform bytecode manipulation. We used the Major mutation framework [123] to generate mutants, and use the mutants as a substitute for real faults. This is a common technique that has been supported by research [124]. We used all the mutation operators provided by the tool. We used Progex [117] to generate CFGs of the source code. The open source projects we used did not have test plans that identified individual test goals. As a proxy, we used the Edge Coverage Criterion (ECC) to specify coverage requirements of the program, and measured the available tests in terms of edges covered. This resulted in test purposes that are valid for our experiment. We used Python scripts to automate the steps in our framework. To flag changes in the source code, we use the `SequenceMatcher` class [120] of the `diffLibPython` library [121].

Although the tools and environment used in this project limit the scope of this framework, most limitations could be overcome by finding or building more robust tools.

## 5.3 An Empirical Study

We carried out two studies to analyze our strategy to make automated tests more efficient.

We start by asking two research questions:

**RQ 5.1** *How much time does the analysis use?*

**RQ 5.2** *How accurate are the analysis results from the framework?*

### 5.3.1 Methodology

We collected two sets of data from four open source code repositories. All projects had JUnit tests. We first collected requirements for statement coverage and edge coverage, and then measured the coverage the tests achieved. We used mutation analysis to simulate changes to the source code, then compared the mutated (*new*) versions of the program to the original version. This gave each test the information needed to decide what action to take when the software was changed.

#### 5.3.1.1 Study One: Preliminary Analysis

We performed a preliminary, small scale study to evaluate how comprehensive and accurate our strategy is.

**i) Subject selection:** The study’s goal was to evaluate the reliability of our approach and the correctness of the automated steps. We used four classes from the Apache Commons Math4 project. All the classes have between 100 to 1000 lines of code and came with between 7 and 74 tests. Table 5.1 gives statistics from these classes.

Table 5.1: Study one: Subject program details of preliminary analysis

Class names	SLOC	#tests	#mutants	EC requirements
DerivativeStructure	637	74	50	283
DSCompiler	885	7	50	436
FiniteDifferences-Differentiator	169	16	50	43
SparseGradientFunction	555	70	50	257
<b>Total</b>	<b>2246</b>	<b>167</b>	<b>200</b>	<b>1019</b>

**ii) Data collection:** We used automated scripts to perform steps 1 (run the tests), 2 (create the CFG), and 3 (create mappings from tests to test requirements) of the test management framework process. The scripts collected statement coverage and edge coverage requirements, and created mappings between tests and which test requirements they



satisfied. We simulated changes to the software under test by using mutation to modify the original program. We used the PiTest mutation testing tool [125] to generate mutants of each Java class. The mutation operators used are shown in Table 5.2. To allow us to perform manual analysis in a reasonable time frame, we randomly selected 25 killed and 25 surviving mutants for each Java class. We placed these 200 mutations  $((25+25) \times 4)$  into the code to create 200 versions of the classes. We created the CFG for each “new” version, and compared them with the CFG of the initial program to identify the changes to the program. The change was then analyzed to check if the test requirements were affected, and in turn, the tests. We then analyzed these changes to identify which action the test need to take in response to the change, which was then presented to the tester on the console.

Table 5.2: Study one: PIT mutators used

Mutator	Example
Boolean false return	<code>return a</code> $\mapsto$ <code>return false</code>
Boolean true return	<code>return a</code> $\mapsto$ <code>return true</code>
Conditional boundary mutators	<code>a &lt; b</code> $\mapsto$ <code>a ≤ b</code>
Empty returns	<code>java.lang.String</code> $\mapsto$ <code>""</code>
Increments	<code>i++</code> $\mapsto$ <code>i--</code>
Invert negatives	<code>return -i</code> $\mapsto$ <code>return i</code>
Math	<code>+</code> $\mapsto$ <code>-</code>
Negate Conditionals	<code>!=</code> $\mapsto$ <code>==</code> , <code>&lt;</code> $\mapsto$ <code>≥</code>
Null return	<code>return a</code> $\mapsto$ <code>return null</code>
Primitive returns	replaces primitive data type return values with 0
Void method calls	removes calls to void methods

**iii) Framework result verification:** We then verified the results for each test and each change by hand, allowing us to identify corner cases that needed to be addressed. We found that our strategy correctly identified which lines were changed and how, and which statements were deleted. In turn, the framework was able to identify test requirements that were affected, and which tests those changes affected. We were also able to correctly flag unsatisfied requirements and failing tests. We also verified the accuracy of decisions

to rerun or delete tests. The results of this verification are presented in Section 5.4; all decisions made by the test framework were correct.

### 5.3.1.2 Study Two: Apache Projects

The goal of our second study was to check the accuracy and comprehensiveness of the framework result, and the time required for the analysis, in a larger project. We used three Apache projects: (1) Commons Math3, (2) Commons CSV, and (3) Apache Commons CLI, for this study.

**i) Subject selection:** To scale up the size of our study, we used the Major mutation testing tool [123] to automate the creation of new (mutated) versions of the software. We switched to Major because it has the ability to export mutated program versions. This change also caused us to switch to Apache Commons Math3 because Major was incompatible with some parts of Math4. Details of our subject program are shown in Table 5.3.

Table 5.3: Study two: Subject program details

Package names	# classes	SLOC	# tests	# mutations	# EC requirements
<b>CLI</b> (1 package)	11	1962	268	516	873
<b>CSV</b> (1 package)	6	1432	205	251	804
<b>math3</b> project (9 packages)					
complex	6	976	159	229	429
dfp	4	2662	36	200	2019
differentiation	5	1708	100	231	971
distribution	27	3560	153	1331	1334
function	8	1173	138	353	288
integration	8	1116	26	380	264
interpolation	14	1767	85	616	862
polynomials	5	741	41	250	419
solvers	14	1650	52	472	701
<b>Total</b>	<b>108</b>	<b>18,747</b>	<b>1263</b>	<b>4829</b>	<b>8964</b>

**ii) Data collection:** As in study one, we used automated scripts to perform steps 1 (run the tests), 2 (create the CFG), and 3 (create mappings from tests to test requirements).

We then used Major to generate the new versions of the software, resulting in 4829 changes to 108 classes. The mutation operators used in this study are shown in Table 5.4. We then created CFGs for each new version and compared them with the CFGs of the original versions. As before, we then analyzed the changes to determine which test requirements were affected, then identified the appropriate action for each test. We recorded the system time for the analysis to answer RQ 5.1. The results of the analysis were shown on the console for the tester to see.

Table 5.4: Study two: Major mutation operators used

Mutation operator	Example
AOR (Arithmetic Operator Replacement)	$a + b \mapsto a - b$
COR (Conditional Operator Replacement)	$a    b \mapsto a \&\& b$
EVR (Expression Value Replacement)	$\text{return } a \mapsto \text{return } 0$
LOR (Logical Operator Replacement)	$a \wedge b \mapsto a   b$
LVR (Literal Value Replacement)	$0 \mapsto 1, \text{true} \mapsto \text{false}$
ORU (Operator Replacement Unary)	$-a \mapsto a$
ROR (Relational Operator Replacement)	$a == b \mapsto a \geq b$
SOR (Shift Operator Replacement)	$a \gg b \mapsto a \ll b$
STD (STatement Deletion)	$\text{return } a \mapsto \langle \text{no-op} \rangle$

**iii) Framework result verification:** We used an automated script to verify results on the 4829 mutants. The script compared the result to the expected result based on the initial program version and the new program version. We found several discrepancies between the result reported by the framework and expected results, as reported in Table 5.6.

## 5.4 Observations and Results

This section presents results from the two studies whose results are shown in Tables 5.5 and 5.6, and our observations from those results. In the preliminary study on four classes from Apache Commons Math4, we checked the test and program status for 50 changes to each Java class. Table 5.5 shows that all program and status decisions in study 1 were correct.

Table 5.5: Result: Study one

Packages	Mutants evaluated	Incorrect results
DerivativeStructure	50	None
DSCompiler	50	None
FiniteDifferencesDifferentiator	50	None
SparseGradientFunction	50	None
<b>Total</b>	<b>200</b>	<b>None</b>

Our second study used 4829 changes to 108 classes, as shown in Table 5.6. We used a maximum of 50 changes per class, drawn randomly from the total number of changes (some classes had fewer than 50 changes). This analysis required a total of 553 minutes, for an average of just over 5 minutes per class. The time required to identify a program change, identify the test status, and determine the appropriate action averaged 6.87 seconds per change. The framework’s recommended action was accurate 94% of the time.

We analyzed each inaccurate result and categorized the reasons into three types.

**a) The program change was not identified correctly.** Our framework only generates control flow graphs for the methods in the class. Therefore, code changes that do not appear in methods, such as changes to statements that declare or instantiate class attributes outside a method, were not identified. This is a limitation of the tool, rather than a problem with the concepts.

This could be solved by adding such information to the control flow graph or by using a different abstraction that represents that information. In our study, 41 of the 290 (14%) inaccurate results fall in this category, that is, 0.8% of the total number of results.

**b) The CFG was not created correctly.** The CFG generation tool we used, Progex [117], was not able to create control flow graphs for some methods in some classes. For example, for code shown in Figure 5.3, graph nodes for statement 6 and 8 were created as independent nodes, as shown in Figure 5.4, instead of connected nodes in one graph. In our study, 89 of the 290 (31%) inaccurate results fall in this category, that is, 1.8% of the total number of results.

Table 5.6: Result: Frequency of test status match and time taken for study two

Projects	Packages	Changes evaluated	Time (mins)	Result match	Result mismatch	% of test status matches	% of test status mismatches
CLI		516	129	456	60	88.37%	11.63%
CSV		251	53	246	5	98.01%	1.99%
math3	complex	229	75	220	9	96.07%	3.93%
	dfp	200	8	199	1	99.50%	0.50%
	differentiation	231	31	226	5	97.84%	2.16%
	distribution	1331	35	1289	42	96.84%	3.16%
	function	353	39	329	24	93.20%	6.80%
	integration	380	35	348	32	91.58%	8.42%
	interpolation	616	91	601	15	97.56%	2.44%
	polynomials	250	24	157	93	62.80%	37.20%
	solvers	472	33	468	4	99.15%	0.85%
<b>Total</b>		<b>4829</b>	<b>553</b>	<b>4539</b>	<b>290</b>	<b>94.00%</b>	<b>6.00%</b>

c) **The line reported by the framework as changed did not match the actual line changed.** The control flow graph considered one statement as one node regardless of the statement length, such as when a program statement spanned multiple lines. For example, some statement spanned four lines in the source code file as shown in 5.5.

Our control flow graph generator placed all four lines into a single node, and identified the entire statement as line 1. However, if a change was made on physical source code line 2 (changing `param[3]` to `param[i]` for example), our tool would match that to line 2 in the CFG, which does not exist in the generated CFG, since lines 2, 3, 4 from the actual source code are part of line 1 in the CFG. In our study, 160 of the 290 (55%) inaccurate results fall in this category, that is, 3.31% of the total number of results.

In summary, the incorrect results were due to relatively minor issues in our tooling, in particular, the CFG generator, not due to conceptual or practical problems with the problem solution.

Thus our answer to RQ 5.1 is that the time taken by the framework to analyze each change averaged 6.87 seconds. This time can be further decreased using code optimization techniques such as reducing the number of file input output operations and optimizing the

database for information storage.

```
1. public static Dfp pow(Dfp base, int a) {
2.     boolean invert = false;
3.
4.     Dfp result = base.getOne();
5.
6.     if (a == 0) {
7.         // Special case
8.         return result;
9.     }
10.
11.    if (a < 0) {
12.        invert = true;
13.        a = -a;
14.    }
15.    return null;
16. }
```

Figure 5.3: Example code of CFG not created



Figure 5.4: Example of incorrect CFG created

```
1 Logistic.value(param[1] - x, param[0],
2                 param[2], param[3],
3                 param[4],
4                 param[5]);
```

Figure 5.5: Example four-line statement

Our answer to RQ 5.2 is that the framework was accurate 94% of the time. Since the incorrect results were due to incorrect or incomplete representation of the program, we are confident the accuracy can be improved even further.

## 5.5 Threats to Validity

We used an open source control flow graph generator tool from github, Progex [117]. The latest release of the tool was in 2019 and the tool did not appear to have been validated. Therefore, problems with the tool could lead to threats to our study. Indeed, we determined that most incorrect decisions the test framework were directly attributable to shortcomings of Progex.

Another potential threat is that all of our classes were obtained from a small number of open source projects. However, the Apache Commons project is large and diverse, with classes that perform many complicated computations.

The code changes were modeled through a fairly simple program abstraction tool, the control flow graph. Result accuracy could be higher if we included data flow information, or a more sophisticated abstraction such as an interprocedural CFG [126] or an interclass graph [127] to capture more details.

Finally, our model of program changes was fairly simple—single order mutants. Although this allowed us to create and analyze thousands of program changes, a more sophisticated model of program changes, or actual changes made to software as documented in changelogs, could provide different insights to this approach to test management.

## 5.6 Conclusions and Future Work

This chapter presents results from a novel, holistic, solution to the problem of managing and evolving automated test suites. As software evolves over time, the suite of automated tests must also evolve. For every change, some tests need to be rerun to verify the change, while other tests are not affected by the change and thus do not need to be run. Further, some tests need to be modified to still run on the modified software, others are no longer relevant and can be ignored, and some new tests need to be created to verify added or modified functionality. These decisions are generally called test management, and despite years of research, test management is still largely done by hand.

Poor test management leads to unchecked growth in the number of tests (test bloat), tests that are no longer correct with respect to the software under test, flaky tests [20,47], and blind tests [22]. Over time, testing becomes more expensive and less effective.

The novel approach presented in this chapter is to give each test the ability to manage itself. To do that, tests need two things. They need to be *self-aware*, that is, know why they exist. Second, tests need to have *self-determination*, that is, be able to choose whether to run, not run, be changed, or be deleted. We have developed a process to support self-managed tests, and a framework that incorporates algorithms and software to automate the self-management approach.

The chapter presents results from an empirical evaluation on open source software, which resulted in two broad findings. First, the *time* needed to create, store, process, and use the information that tests need to manage themselves was quite reasonable. Second, the *accuracy*, in terms of whether tests made correct decisions, was quite high, with the primary limitation stemming from the capabilities of the control flow graph generator that we used.

### 5.6.1 Future Work

In the future, we hope to extend these ideas in several ways. As currently configured, when a test discovers that it needs to change to accommodate changes in the software under test, it alerts the test for manual intervention. We believe that automatic repair [106] techniques could be used to automatically update some tests. The scope of change is smaller than for general software, so the potential for success may be higher for automated tests. We can also investigate and address scalability to ensure this approach works for larger programs, and applicability, to ensure this approach works for real programs and can be used in practice.

More broadly, our demonstration implementation works in the context of test structural requirements derived from control flow graphs. We can apply this approach to other types of test requirements, such as those based on functional or non-functional software requirements. We can also evaluate the framework with real software faults, and to further



investigate how test requirements can be better captured and maintained as the program evolves. Finally, since our research is attempting to automate work that is currently done mostly by hand, future work can make a direct measure of improvement by comparing results from our process with results from a strictly human process.

## Chapter 6: GUI-based Test Oracle Automation - Magneto

### 6.1 Introduction

Automation in software testing has three main types: *test input generation*, that is, generating test input values automatically; *test execution*, that is, executing tests automatically; and the *test oracle*; that is, checking whether a test has passed (no fault was found) or failed (a fault was found). Automatically checking whether a test passes is particularly challenging, largely because it requires understanding nuances of the expected program behavior to determine when deviations occur [34]. Thus, designing effective test oracles can be particularly challenging for certain software domains where application behavior is diverse and difficult to model.

Mobile applications (or *apps*) are widely used [128, 129], and often failure prone [130]. Thus, practitioners and researchers are developing automated techniques to improve how they are tested [131–137]. However, automatically testing mobile apps has proven to be quite difficult. A crucial limitation of nearly all automated testing techniques for mobile apps is related to test oracles—most generated tests use only *implicit oracles*, such as crashes or raised exceptions. Researchers have found that only around 1/3 of all failures result in a crash [1, 138], which means that many tests that rely on implicit oracles are, in effect, blind. Thus, failures are not reported, even when tests find them.

Automating test oracles for common types of mobile app failures can advance the progress in automated software testing, ranging from crowd-sourced testing (e.g., through automated identification of faults from crowd app usages) to regression testing (e.g., through automated generation of test assertions). Automated test oracles could also help in tasks other than testing. For instance, they may also help developers take better advantage of automated failure reproduction techniques [139, 140] (e.g., by using the oracles to confirm

that a failure was reproduced), and cope with rapidly evolving platforms and APIs [141–146] (e.g., by detecting cross-platform inconsistencies).

Test oracles are hard to automate for mobile apps partly because they are event-driven and GUI-based, thus it is inherently challenging to automatically reason about their behavior. Past work on automating GUI test oracles has tried to build formal models of GUI-based software [97], infer useful oracles from existing test cases [147], generate test cases with oracles based on common functionalities [96] or visual glitches in games [148], and by fuzzing user interfaces [83]. However, these approaches have tried to either exploit information from existing test cases, which are often not available for mobile apps [149], or try to *infer* correct program behavior, which is imprecise and often inaccurate.

This chapter presents two novel contributions to overcome these limitations. First, we systematically categorize the characteristics of common non-crashing, functional, failures in Android apps to better understand how effective automated oracles can be designed. Second, we automate oracles for some common non-crashing failures in mobile apps by automatically detecting empirically derived patterns in the GUI and checking for the presence of a failure. We use a combination of natural language processing and computer vision techniques to avoid incorrectly identifying failures.

For our first contribution, we systematically designed the first *behavioral oracle taxonomy* for Android apps that identifies and categorizes common GUI element behaviors, expected app responses to GUI interactions, and corresponding failures. This novel taxonomy is based on 124 reproducible bug reports extracted from a reports dataset [138] mined from open source apps on GitHub. The taxonomy was derived using an open coding process [150] following guidelines from constructivist grounded theory [151], and involved multiple rounds of independent, multiple-author labeling with regular norming meetings. The resulting taxonomy describes hierarchical properties of Android app failures, starting with how they manifest through the GUI, which illuminate *GUI-based* patterns that we then use to automatically detect common types of failures.

For the second contribution, we designed an approach, called MAGNETO (autoMAtinG

aNdroiD tEsT Oracles) to automatically detect a set of the failure patterns identified by our taxonomy, starting with the most common patterns. We use both computer vision and natural language processing techniques to detect failure patterns in mobile app user interfaces. We implemented this approach for five of the oracle types derived from generalized failure behavior patterns and discuss advancements that would be required to implement additional oracle classes as promising directions for future research on test oracle automation. To evaluate the applicability of our approach for creating automated test oracles, we applied it to three additional real failures we selected for each of our five automated oracle types (resulting in 15 failures). Our results indicate that our oracles effectively revealed 11 of the 15 failures in the faulty versions of the apps, while reporting only one false positive<sup>1</sup>. Our approach also uncovered two previously unknown failures in two of the apps. Additionally, we combined our approach with a recent automated testing tool for Android apps to examine its practical effectiveness. Our oracles found two failures uncovered by the tool that would have otherwise gone undetected, with an extremely low false positive rate. Finally, we explain how the failures that were **not** detected by our test oracles could be revealed with additional engineering effort.

In summary, this chapter makes the following contributions:

- The first *GUI-based behavioral oracle taxonomy* for Android apps that describes the characteristic patterns of different types of failures that have occurred in Android apps “in the wild.”
- An automated approach to detect failure patterns in mobile app GUIs for five of our identified oracle types.
- A comprehensive empirical evaluation of our five automated oracle patterns to demonstrate generalizability, and compatibility with an automated input generation (AIG) tool.
- A comprehensive replication package and online appendix that includes our complete behavioral oracle taxonomy, the source code of our automated oracle approach, and all of the data used to build our taxonomy and carry out our evaluation [152].

---

<sup>1</sup>In the context of our test oracles, a *false positive* occurs when the test oracle thinks the software failed when it behaved correctly.

We discuss the derivation methodology and results of our *behavioral oracle taxonomy* in Section 6.2. Section 6.3 describes our automated approach, MAGNETO. Our empirical evaluation methodology and results are discussed in Section 6.4. Section 6.5 discusses findings and limitations of MAGNETO and Section 6.6 concludes the chapter.

## 6.2 Behavioral Oracle Taxonomy

We develop a *behavioral analysis* of real failures to understand the types of generalized oracles that may be applicable to Android applications. By understanding how app failures both affect users and manifest through the GUI, we can extract common patterns of *correct and expected* behavior that could be detected and checked during automated testing. While prior work has introduced taxonomies of Android faults [80, 153, 154], they were primarily concerned with identifying root causes, interaction features, or presentation failures. This is distinct from our focus on the GUI-based behavior of software failures.

This section describes our oracle taxonomy, its novelty, and our taxonomy derivation methodology. Our taxonomy consists of a classification of 124 reproducible Android bug reports at four different abstraction levels: ( $\mathcal{L}_1$ ) a *user-facing* characterization of the failures (effects on user), ( $\mathcal{L}_2$ ) a characterization of the *failure manifestation* as observed through the GUI, ( $\mathcal{L}_3$ ) the *resources* required to construct an oracle for the studied failures, and ( $\mathcal{L}_4$ ) *behavioral invariants* that characterize common patterns of expected behaviors. Our taxonomy characterizes existing failure behaviors in Android apps. It also synthesizes common patterns of generalized expected behavior that could be used in automated oracle construction. Hence, we refer to it as the *behavioral oracle taxonomy*.

### 6.2.1 Taxonomy Derivation Methodology

Our taxonomy derivation process consists of two main steps:

**Step 1: Resource collection.** To create our failure taxonomy, we needed a large, diverse set of *fully-reproducible* bug reports as evaluators must be able to analyze the expected and observed behavior of the bug to create an accurate behavior characterization. We used

the ANDROR2+ dataset [138], which is the largest dataset of crashing and non-crashing bug reports for Android apps. ANDROR2+ contains automated reproduction scripts for 180 bug reports from 53 open source apps on Google Play and GitHub. We analyzed the 124 non-crashing bugs included in the dataset. To aid qualitative open-coding, we ran the automated reproduction scripts from ANDROR2+ on an emulator and captured the event traces using the Linux `getevent` utility, and used `uiautomator` to collect sequences of screen metadata, app screenshots, and videos for each bug. The GUI metadata contains a run-time specification of the organization and properties of the GUI components from a screen in XML format, which we used to characterize how failures were displayed.

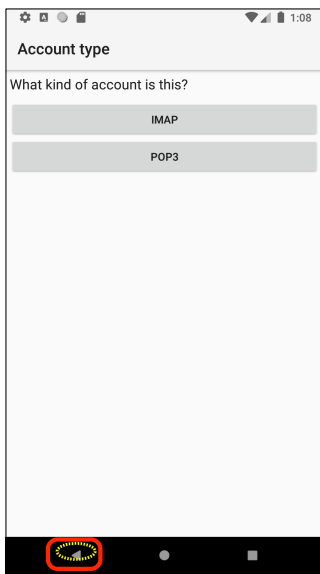
**Step 2: Qualitative analysis.** To build up the knowledge base of our taxonomy, we conducted a qualitative study using both open and axial coding [151]. Open coding involves describing and labeling data, whereas axial coding involves identifying relationships between previously coded labels.

We performed open coding of failures in three steps. First, we developed a codebook to describe the initial labels for the taxonomy categories, their definitions, and labeling rules, consistent with accepted methodology [151]. We then used open coding to derive and assign the labels for the first three parts of our taxonomy: ( $\mathcal{L}_1$ ) the failure’s *Effects on Users*, ( $\mathcal{L}_2$ ) the *Failure Manifestation*, and ( $\mathcal{L}_3$ ) the *Resources Required for Oracle*. Then, given the derived labels and categorizations, we used axial coding to derive the ( $\mathcal{L}_4$ ) *app Behavior Invariants* [151].

Figure 6.1 illustrates an example bug description, and screenshots for Bug#3971 from the K9Mail application. Figure 6.2 provides a high-level overview of our taxonomy categories, which we describe in detail using the example in the next subsection.

## 6.2.2 Taxonomy Results

Figure 6.3, Figure 6.4, Figure 6.5, and Figure 6.6 illustrates our GUI-based oracle taxonomy that categorizes the *behavioral patterns* of non-crashing Android bugs. The four figures (Figure 6.3, Figure 6.4, Figure 6.5, and Figure 6.6) illustrates one of each taxonomy category



App failure point

**Bug Title:** App Closes on Pressing Back Button in Manual Setup

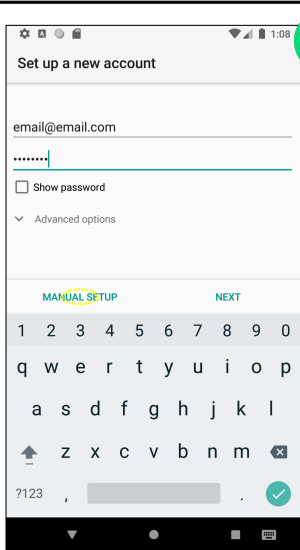
**Bug Description:**

Expected Behavior: App should go back to the previous page

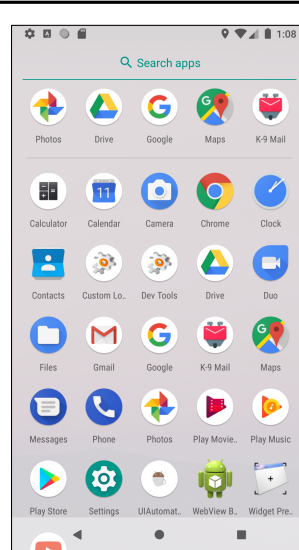
Observed Behavior: App closes when back pressed

Reproduction Steps:

1. While creating a new account, click manual setup
2. Press back button
3. App closes



Expected behavior



Observed behavior

Figure 6.1: Illustration of bug #3971 from the K9Mail app

(which also match the overview in Figure 6.2). We provide complete results and all data in our online appendix [152]. Next, we describe our derived taxonomy structure in detail:

$\mathcal{L}_1$ . **User-facing Failure Type (Effects on Users):** A failure can have different impacts

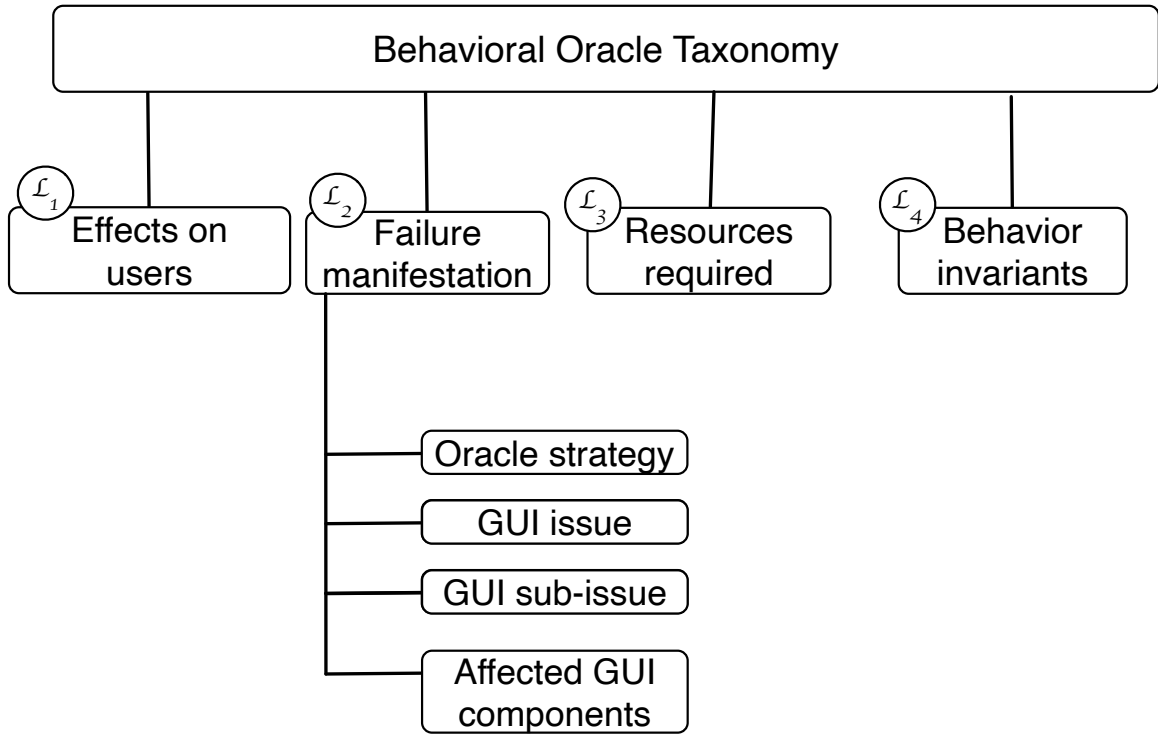


Figure 6.2: Oracle taxonomy categories

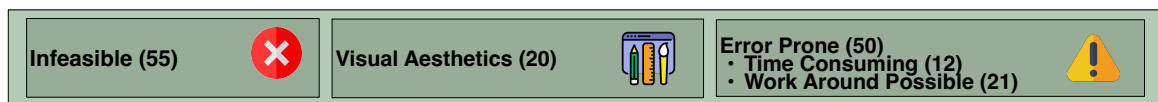


Figure 6.3: ( $\mathcal{L}_1$ ) Categorization of failure effects on users



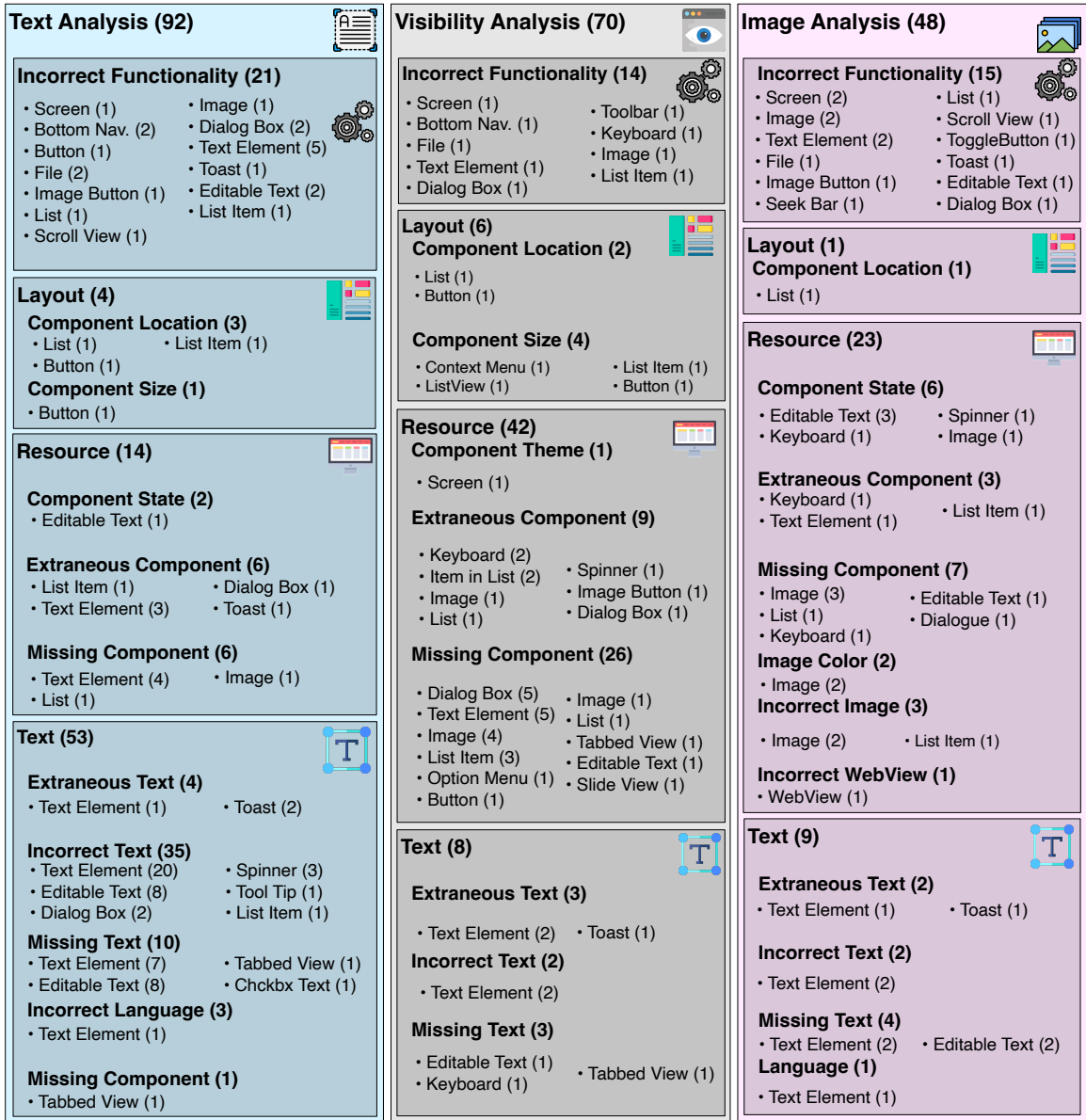


Figure 6.4: ( $\mathcal{L}_2$ ) Oracle analyses and failure categorization

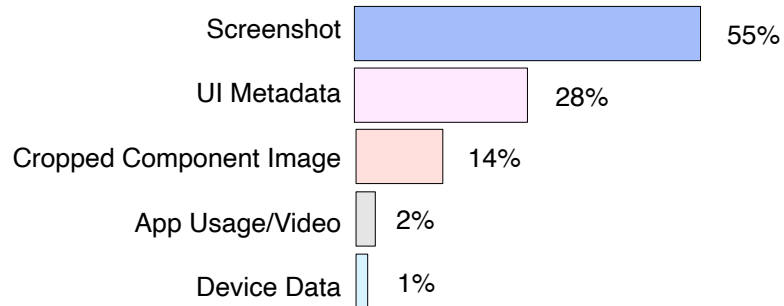


Figure 6.5: ( $\mathcal{L}_3$ ) Percentage of resources required for automated oracle

- 1) User-entered data should display correctly (12) ✓
- 2) Action name and action should match (10)
- 3) Theme change should be reflected on all screens (9) ✓
- 4) The screen content should not change on rotation (6) ✓
- 5) Dialog Box should only appear after certain actions (5)
- 6) GUI elements should not be missing (5)
- 7) The language change should be reflected on all screens (4) ✓
- 8) Pressing back should lead to the previous screen (4) ✓
- 9) Padding of items in List should be consistent (3)
- 10) Duplicate elements should not be present (3)
- 11) Element size should match expected size (3)
- 12) Certain GUI elements should only appear in pairs (3)
- 13) App state should (not) restore after certain actions (3)
- 14) Cursor Position should be consistent with user input (3)

Figure 6.6: ( $\mathcal{L}_4$ ) App behavior invariants derived from the taxonomy. ✓ denotes oracle implemented invariants.

on the users' ability to complete a task. For instance, a failure could prevent the task from being completed, or simply delay completion. Labels were not pre-assigned for this category as we were discovering patterns.

We defined three mutually exclusive labels (and two sub-labels) as shown in Figure 6.3 to categorize a failure's impact on a user:

$\mathcal{L}_{1.1}$  *Infeasible* is used when a user cannot complete a task because of the failure. For example, if a user wants to submit a form but the submit button is missing, the failure makes the task infeasible. For our example in Figure 6.1, the user was unable to complete a task making it *infeasible*.

$\mathcal{L}_{1.2}$  *Error prone* means users are more likely to make mistakes because of the fault. For example, assume a form has a red button for submit and green for cancel. Since green buttons usually mean submit and red means cancel, users are more likely to make a mistake when the colors are swapped. Error prone faults are further defined by two sub-labels:

$\mathcal{L}_{1.2(a)}$  *Work around possible*: Users can complete their task, not with the sequence of actions they intended, but with different, often more difficult, actions. Consider the missing submit button discussed above. If the submit button appears only when the app is in landscape mode, the user can work around the problem by rotating the screen.

$\mathcal{L}_{1.2(b)}$  *Time consuming*: Users need to repeat actions or need longer to complete the task *e.g.*, a submit button needing to be pressed multiple times.

$\mathcal{L}_{1.3}$  *Visual aesthetics* means that users can complete the task but the GUI is aesthetically unappealing. For example, this may occur when two buttons on a form are not aligned.

Our taxonomy illustrates that the vast majority of our studied bugs can be categorized as either *infeasible* (44%) or *error prone* (40%). This means that 84% of our studied bugs either prevent a feature from being exercised or make the user more likely to make a mistake.

Table 6.1: Oracle strategy labels

Oracle Strategy	Summary
Color analysis	Check if component color is as expected
Data analysis	Check if displayed text matches data from another app
Image analysis	Check if image on screen matches expected image
Position analysis	Check if component position matches expected position
Size analysis	Check if component size matches expected size
State analysis	Check state of component property like required, enabled, selected
Text analysis	Check if text on screen matches expected text
Theme analysis	Check if screen theme matches expected theme
Visibility analysis	Check if component is visible on screen
Webview analysis	Check if screen (image, text) matches expected screen

$\mathcal{L}_2$ . **Failure Manifestation:** This category characterizes how the failure is displayed to the user. It has four sub-categories: (a) oracle strategy, (b) GUI issues, (c) GUI sub-issues, and (d) affected GUI components. Figure 6.4 shows a snippet of failure manifestation categories. It groups the *Failure Manifestation* categories by *Oracle Strategies*. The nested boxes contain information related to the *GUI Issue*, *GUI Sub-Issue*, and *Affected GUI Components*.

$\mathcal{L}_{2.1}$  Oracle Strategy: This category identifies general strategies that could detect the failure from the bug report. We have 10 labels in this category. The oracle strategy labels and their summary is shown in Table 6.1.

We show results for three of the oracle strategy labels (text analysis, visibility analysis, and image analysis) in Figure 6.4. Results for the remaining oracle strategy labels are shown in Appendix A.2. These labels specify which element or element behavior needs to be verified in the test oracle. Our example bug report has two labels assigned, *Image Analysis* and *Visibility Analysis*, meaning the faulty behavior could be detected through either the presence or absence of the expected screen (visibility) or through a similarity analysis (image analysis) of the observed and expected screens.

$\mathcal{L}_{2.2}$  GUI Issues: This category describes the GUI failure at a high level. Initial labels were assigned starting from the GUI presentation issues derived by Moran et al. [154], and refined throughout the coding process to include functional issues that this pre-existing taxonomy did not describe:

$\mathcal{L}_{2.2}$ (a) *Incorrect Functionality*: The failure exhibits incorrect logic, such as a form that does not save user info on submit. Our example from Figure 6.1 falls into this category due to the unexpected state of the app, as opposed to an issue with an individual GUI component.

$\mathcal{L}_{2.2}$ (b) *Layout*: The failure manifests through the layout of components such as positioning and size. For example, a form that displays unaligned buttons is a layout issue.

$\mathcal{L}_{2.2}$ (c) *Resource*: Failures that manifest as incorrect elements such as a missing button, a wrong image, etc.

$\mathcal{L}_{2.2}$ (d) *Text*: The failure is associated with the text content of the GUI. For example, a form that displays “submit” instead of “cancel” in a form.

These labels are not completely mutually exclusive. For example, a bug with a faulty “paste” button can be labeled *incorrect functionality* because it is caused by incorrect app logic. It can also be labeled *missing component* because a component is missing after the “paste” action.

We found that the *resource* issues (30%) are the most common GUI issue with the *text* issues (27%) second. 50% of the *resource* issues were *missing component* and 80% of the *text* issues were *incorrect text*.

$\mathcal{L}_{2.3}$  GUI Sub-Issues: This categorization refines the labels of the previous sub-category and so depends on its labels. We define the following labels nested under the GUI issues. *Layout* is divided into two sub-issues: (a) size, and (b) position. *Resource* is sub-categorized into eight sub-issues: (a) missing components, (b) extraneous components, (c) component state, (d) image color, (e) incorrect image, (f) element type, (g) element shape, and (h) element theme. *Text* is sub-categorized into

four sub-issues: (a) font color, (b) font style, (c) incorrect text content, and (d) incorrect language. *Incorrect Functionality* has no sub-categories. Our example does not have a sub-categorization, as the *Incorrect Functionality* label was not further refined.

**$\mathcal{L}_{2.4}$  Affected GUI Components:** This category specifies the failing GUI components.

We initialized labels for this sub-category by drawing upon past work in reverse engineering GUIs [155], and expanded these using terms from UI design [156]. We derived 28 different component types, and also observed that a fault can have multiple failing GUI components. Further, these 28 components are independent of oracle strategy, GUI issues, or GUI sub-issues categories described earlier. The affected GUI component in our example bug is the *Back Button*, or more simply, the navigation bar. Also, since the resulting screen was also incorrect, the *Screen* label is assigned.

We found that static *Text Elements* are the most common GUI element that fails. 58 of 192 failing components are *Text Elements*, with the next most common being *Editable Text*.

**$\mathcal{L}_3$ . Required Oracle Resource:** This category lists resources needed for automated oracle implementation. We created five labels that describe resources needed for oracles: *screenshots* (55%), *UI metadata* (28%), *cropped GUI component images* (14%), *video* (2%), and *device data* (1%). The fault in our Figure 6.1 example can be detected through either image or visibility analysis, and thus, we need the resource *full-screen app Screenshot*. We did not assign initial labels to this category. Five authors independently analyzed the bug reports using the resources collected previously, using the category definitions listed above, using negotiated agreement [151] to iteratively resolve inconsistencies among the assigned labels.

**$\mathcal{L}_4$ . App Behavior Invariants:** While our characterization of failure patterns grounded in the user interface provides a useful overview of common *individual* app failure points and patterns, our main goal is to derive *generalized* failure behavior patterns that could

reliably be detected during automated testing. To accomplish this goal, we performed axial coding to derive a set of generalizable *invariant behaviors* for the failures. We generalized the expected behaviors to be tied to user actions but independent of app logic. We call such user actions *triggers*. Given a trigger, we expect the associated app behavior invariant to hold across multiple apps. These invariants represent common usage patterns and expected GUI behaviors resulting from these patterns. For example, the invariant behavior derived from our example bug report is “*pressing back should lead to the previous screen,*” which was observed across several bug reports. Figure 6.6 shows the 14 app invariants we identified from the taxonomy.

### 6.2.3 Behavioral Oracle Taxonomy Insights

This section discusses important insights derived from our taxonomy, and comments on its future applicability to research and practice on automated testing for mobile apps. At least 3 broader principles have emerged from the taxonomy:

1. ***In mobile apps, non-crashing bugs can be categorized into varying severity levels based on their user-facing effects.*** We discovered that 44% of the bugs stop users from completing a task and 56% reduced the user experience. This finding, and the distribution of bugs across failure categories, helps motivate the need for additional work on detecting, understanding, and prioritizing bugs during the testing, bug localization, and bug fixing processes.
2. ***Despite their variable, event-driven nature, there are common invariant behaviors that, when deviated from, can signal a potential failure.*** We derive 14 app behavior invariants, which can be used in future work to construct new types of automated oracles as machine learning and computer vision techniques improve.
3. ***There are a common, and fairly consistent, set of manifestation patterns of bugs through both individual and groups of UI components.*** We identify the manifestation patterns of Android bugs through specific UI components, which

has several implications for future app testing techniques, which we elaborate upon below.

In addition to these high-level insights, our taxonomy is positioned to enable new software engineering techniques or enhancements to existing techniques. For instance, it can inform the creation of new static analysis tools that preemptively warn developers against bugs exposed through the GUI that may hinder or block features from users. Our identified bug manifestation behaviors could be used to prioritize crowd-sourced testing tasks in an effort to reveal more bugs with a smaller amount of effort, or delegate certain bugs toward detection via automated tools. Finally, the prevalence of the different types of failures could be used to help prioritize UI test cases for mobile apps.

### 6.3 MAGNETO: Automating Android Test Oracles

This section describes our automated approach, MAGNETO (autoMAtinG aNdroid tEsT Oracles), for revealing mobile app failures.

#### 6.3.1 Overview

Figure 6.7 depicts the workflow of MAGNETO. It has three main phases: (1) *App Execution*, (2) *Oracle Trigger Detection*, and (3) *Automated Oracle Execution*.

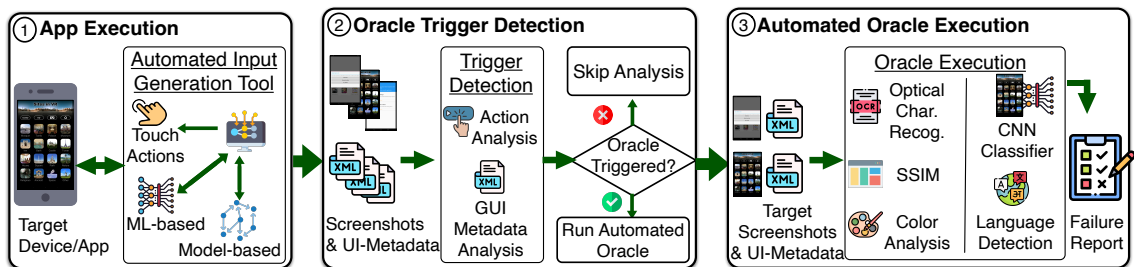


Figure 6.7: An overview of the workflow of MAGNETO



### 6.3.1.1 App Execution

MAGNETO is meant to be used in conjunction with automated input generation (AIG) tools for Android. As such, this first step involves executing a chosen tool, and collecting the output that MAGNETO uses to examine the generated test, apply its automated oracles, and determine whether a failure occurred. To bolster the practicality of our approach, MAGNETO operates on data that is commonly generated by, or can be easily collected by, current AIG Tools for Android apps. That is, MAGNETO requires run-time execution information before and after each test input that is generated by an automated testing approach. More specifically, for each test input it requires: (i) the touch action performed (e.g., tap, swipe), (ii) the location of the action, (iii) screenshots before and after the action occurred, and (iv) GUI metadata that describes the current UI layout hierarchy (collected via `uiautomator`), before and after each action. We specify an extensible JSON format that captures this information in listing A.3 in Appendix A.

### 6.3.1.2 Trigger Detection

Unlike traditional software where app state can be checked at the end of a sequence of user actions, mobile app states need to be checked after every user action [97]. This creates a lot of GUI states to verify, adding to the expense, and increasing the risk of false positives. Thus, we run the automated oracle every time there is a specific user action, which we call a *trigger*. A *trigger* is a user action that results in a particular app behavior. For instance, pressing the `back` button should change the current app screen to the previous screen. Here the `back` button is a *trigger* and the transition to the previous screen is the expected behavior.

### 6.3.1.3 Oracle Execution

Once we detect a *trigger*, we identify the *app behavior invariants* associated with it from the behavioral oracle taxonomy.

We prioritized implementing invariants that are: (i) prevalent ( $\geq$  three instances) in our

taxonomy, and (ii) amenable to current unsupervised computer vision and program analysis techniques. Thus, we developed five type of oracles to verify five app behavior invariants. They are listed and described in Section 6.3.3.

We did not develop oracles for app behavior invariants that were derived from relatively few failures or required information from the bug report to verify. We discuss promising avenues of future work for such cases in Section 6.6. Using the same taxonomy, we identify and execute the appropriate oracle for the app behavior invariant. When executed, the oracle either reveals the failure or marks the app’s behavior as being correct.

### 6.3.2 Oracle Implementation Details

We used six techniques to compare expected and actual screens in our automated oracles.

1. *Structured Similarity Index Measure* (SSIM): We used SSIM to calculate perceptual differences between images [157]. It takes two input images of the same pixel size and quantifies the structural similarity between them. SSIM’s output value is in the range of -1 to 1; -1 means they have no similarity and 1 means they are identical. This measure applies because app states are captured visually in mobile apps.

We calibrated the SSIM threshold by hand comparing representative image pairs with varying levels of known structural similarity. We found that small differences in places like the status bar state and the bottom navigation bar reduced the SSIM even when the screens were the same by our judgement. We also found that the SSIM was less than 1 if the content updated on reload. Conversely, we found that two completely different screens from the same app were not scored as -1 because of structural similarities in the screens. Based on this analysis, we set our threshold to be +0.8, that is, when the SSIM is less than +0.8, the screens are considered to be structurally different.

2. *Delta E*: **Delta E** measures change in perception of two colors. It was originally introduced by the International Commission on Illumination (CIE) to address the problem of measuring perceived color difference [158]. We use it to decide if two colors can be considered equivalent.

`Delta E` values range from 0 to 100, with lower values indicating better matches. Typically, two colors with `Delta E` value less than 2 is considered to be perceptually equivalent [159,160], thus we use 2 as our threshold.

3. *Text detection:* We use the optical character recognition tool `Python-tesseract` [161] to identify text embedded in app screenshot images. We first identify the text bounding boxes and then the tool reads the text in the bounding boxes.
4. *Natural language detection:* After detecting the text in the previous step, we use `Polyglot` [162] to detect the language used.
5. *Image classification:* We use binary image classification technique to identify if a screen has a GUI keyboard visible on the screen. SSIM value of the same screen with and without a GUI keyboard is different. Text detection techniques when run on screen with visible GUI keyboard detects the keyboard letters as well. Comparing SSIM or detected text from screens with and without keyboard can lead to incorrect result from automated oracle. To avoid such error, we trained a Convolutional Neural Network [163] model using the images with and without keyboards from the Rico dataset [164]. We then use this model to decide if a screen has a keyboard. This model performs extremely well (*i.e.*, >98% accuracy).
6. *Image theme detection:* We consider the image “theme” to be the dominant color in the image. We identify the theme by clustering the pixel colors using the k-means algorithm [165]. Once we have clustered the image pixels, we consider the dominant color to be the centroid of the largest k-means cluster, and consider that color to be the theme.

### 6.3.3 Oracle Descriptions

This section describes the five oracles we designed. For each, we describe the expected behavior, the user action that triggers the oracle’s execution, and details of what the automated oracle checks.

#### **Oracle 1: The screen content should not change on rotation.**

*Trigger:* The user rotated the screen.

*Expected behavior:*

- User input should not disappear when the screen rotates.
- The displayed text and images should not disappear when the screen rotates.

*Oracle details:*

We detect changes in screen orientation from the rotation value in the UI metadata XML. If the values in user input, or image view fields are missing after orientation change, a failure was revealed. If  $>0.5$  of the text view is missing, a failure was revealed. We set this threshold to account for text view elements that might be visible only after scrolling.

**Oracle 2: User-entered data should display correctly.**

*Trigger:* Text buttons, image buttons, or text views with text such as “set”, “done”, “save”, “ok” pressed.

*Expected behavior:* User-entered data should be displayed in the screen following saving action.

*Oracle details:*

We check if there was a previous “save” or “cancel” trigger and collect inputs from text fields, check boxes, and radio buttons on all the screens since the last trigger was executed. We verify if the screen shown after the trigger displays the user-entered data.

**Oracle 3: Pressing back should lead to the previous screen.**

*Trigger:* Execution information has a GUI element with idXML BACK\_MODAL, or the user tapped the device screen on the left end of the bottom navigation bar.

*Expected behavior:* The previous screen should be displayed.

*Oracle details:*

We compare the previous screen with the one after the *back* action using SSIM. We cut out the status and bottom bars to reduce the noise in SSIM. Images with keyboards can also skew the SSIM result so we identify the images with a keyboard using our image classification approach and use the same image without the keyboard.

If the SSIM value of the two screens is  $\leq 0.8$ , we mark the screens as structurally different. This means the back button did not go to the previous screen, thus a failure was revealed.

Otherwise, no failure was found. Since some apps have high structural similarity in all their screens, we analyze the displayed text when the SSIM is  $>0.8$ . For text comparison, we use the text detection approach from Section 6.3.2 to detect text in the app screenshot. Once identified, we compare the fraction of mismatch between the two screens. If the mismatch between expected text and actual text on screen is  $>0.5$ , the screens are flagged as different, thus a failure was revealed.

**Oracle 4: The language change should be reflected on all screens.**

*Trigger:* User clicked on a component with text “language” or opened a window with keyword “language” in its title and activity name.

*Expected behavior:* The user-selected language should be applied to subsequent screens.

*Oracle details:*

We find the selected language from the execution information, which contains details of the components the user interacted with. Next, we detect the text on the screenshot image after language is set. We use `Polyglot` to detect if the new language matches the selected language. If  $>.05$  of the total text on the screen does not match selected language, a failure was revealed. We set this low threshold because we observed that language failures sometimes occurs for individual words.

**Oracle 5: Theme change should be reflected on all screens.**

*Trigger:* Execution information has a GUI element with `idXML BACK_MODAL`, or the user tapped the device screen on the left end of the bottom navigation bar.

*Expected behavior:*

- A theme set by the user should be applied successfully.
- The selected theme should apply to subsequent screens.
- The text view should not disappear as a result of theme change.

*Oracle details:*

We compare themes from two screenshots—before and after the theme was set. We assign the themes using the image theme detection technique from Section 6.3.2. Given two themes, we use `Delta E` to decide if they match. As described in Section 6.3.2, if

Delta E is less than 2, the theme colors are perceptually equivalent and we can conclude that the themes match. If the themes before and after the change match, a failure was revealed. If theme was applied successfully, we consider the new theme to be as expected. We then compare all the subsequent screens in the execution trace to the expected theme using Delta E. If they do not match, a failure was revealed. We also check to see if the new theme obscures any text. We compare text on the before and after screens. If the match is  $\leq 0.5$ , most text is obscured and a failure was revealed. We set the threshold at 0.5 because some text might intentionally be hidden.

## 6.4 Empirical Evaluation

This section discusses our empirical evaluation of MAGNETO. In the evaluation, we target the following research questions (RQs):

**RQ 6.1:** *How accurate is MAGNETO's oracle trigger detection?*

**RQ 6.2:** *What is MAGNETO's success rate in detecting failure?*

**RQ 6.3:** *How reliable is MAGNETO in signaling failures?*

**RQ 6.4:** *Can MAGNETO be combined with existing AIG tools?*

**RQ 6.5:** *How does MAGNETO compare with existing work that detects UI Display Issues?*

### 6.4.1 Study Context

To evaluate MAGNETO, we identified and manually reproduced new bug reports that were not considered when deriving the behavioral oracle taxonomy. We evaluated with new faults to avoid bias from evaluating on the same faults we used to derive the oracles. We identified a set of bug reports that are in the scope of expected behaviors we can verify with the automated oracles. We then tried to reproduce the failures in the bug reports to include them in our evaluation data set. Finally, we ran the automated oracles on the evaluation data and report the number of failures revealed, and the number of false positives that occurred.

### 6.4.2 Methodology

To derive our evaluation data-set, we started with 6365 closed bug reports from open source Android apps made available alongside the data related to 180 fully-reproduced bug reports in ANDROR2+ [138]. Note that this list of reports is orthogonal to the set of 180 fully-analyzed bugs, and was meant to serve as a resource for extending the dataset. We performed an initial filtering of bugs using label studio [166] to remove those that clearly did not match our implemented oracles, resulting in 1145 labelled bug reports from the original 6365. Then, based on further manual filtering by reading bug descriptions, user comments, and developer comments, we were left with 163 bugs that, based on their description, we were able to reasonably conclude exhibited behavior similar to that expected by our automated oracles. This filtering is important because our oracles are tied to a specific behavior invariant. For instance, language oracles can only detect failures occurred when setting language. Of the 163 bugs selected, we randomly selected bug reports for reproduction until our filtered bugs were exhausted.

We then reproduced each of these randomly selected bugs to determine whether they actually implemented a fault that matched our oracle definition. The failure reproduction process starts by collecting the faulty app version associated with the bug report, failure reproduction steps, and the corrected version of the app. We then attempted to reproduce the bug by installing the faulty version of the app in an Android emulator, following the steps in the description.

We required app execution data to carry out our evaluation (see Section 6.3) for action sequences that reproduced our collected bugs. Instead of using an automated testing tool to generate inputs that may or may not exercise the faults, we opted to control the process and manually executed the failure reproduction steps in the faulty and fixed app versions and collected the screen UI XML, app screenshots, and execution information. This process was also time-consuming, as we had to bugs if we could not reproduce the failure or could not collect the resources needed to automate the test oracle.

A significant challenge in this study was that we used real faults from real mobile

apps. This increases the validity of the findings, but with the trade-off of requiring a large amount of manual work, because (i) many bug reports on GitHub are not reproducible, (ii) reproduction steps in the bug description are often missing or vague [167], and (iii) the faulty or fixed versions of the apps are not always available. The manual work involved in all the steps, including collecting the app versions, verifying that the apps are executable, reproducing the failures, and collecting the resources, required several weeks for two authors to finish. At the end of this process, we had complete evaluation resources for 15 unique faults, three each for our five expected behaviors.

MAGNETO uses data that is commonly generated by AIG tools for Android apps such as Stoa [168], APE [131], etc. To demonstrate generalizability on a larger dataset beyond our 15 ground truth bugs, and further analyze the potential for MAGNETO to find bugs or trigger false positives, we execute MAGNETO on the output generated by the state-of-the-art tool APE [131]. We ran APE on the data set used to evaluate the tool in its original paper. Using the artifact made available by the authors, and following their recommended emulator settings, we were only able to run APE on 4 out of 15 apps from the author’s original dataset for 30 minutes. This was primarily due to emulator lock-ups. We then ran MAGNETO on APE’s output for the four apps.

There has been recent work in the software engineering research community on using machine learning techniques to detect UI display issues [169–171]. While such tools are not intended to detect the type of behavioral anomalies targeted by MAGNETO, given that they do detect UI issues, we aimed to determine whether such tools can detect any of our oracle triggers or the underlying bugs. As such, we compare MAGNETO with the GUI display issue detection tool OwlEyes [169], as it functions on single screenshots, and is publicly available. We trained OwlEyes on the dataset provided by the authors of OwlEyes, and applied it to detect UI issues in the screenshots collected from our 15 ground truth bugs.



### 6.4.3 Evaluation Metrics

We reproduced the failures and collected resources (screen UI XML, screenshot images, execution information) as described in Section 6.3.1.1. Then, we executed the automated oracles on the collected data to answer the research questions.

To evaluate the accuracy of MAGNETO’s trigger detection technique (**RQ 6.1**), we identify two possibilities: whether the trigger was *present and detected* (TP - true positive) and whether the trigger was *not present and not detected* (TN - true negative). Recall that a *trigger* is a user action that results in a particular app behavior. A trigger may or may not cause an app failure. Note that each failure has one trigger (e.g. faulty back button), however, several valid *benign* triggers may exist within the execution data that do not lead to a failure (e.g. a back button clicked that indeed moved the user back a screen).

We measure MAGNETO’s trigger detection accuracy as:

$$\text{trigger accuracy} = \frac{\text{trig. present detected (TP)} + \text{trig. not present not detected (TN)}}{\text{total expected triggers}}$$

We answer **RQ 6.2** by checking if MAGNETO revealed a failure when there was a failure.

We calculate the recall as:

$$\text{Magnetor recall} = \frac{\text{revealed failures (TP)}}{\text{total expected failures (TP + FN)}}$$

We answer **RQ 6.3** by checking if the oracle incorrectly flagged a failure when there was no failure. We measure this by calculating the number of false positives as:

$$\# \text{ false positives} = \frac{\text{incorrect failure label (FP)}}{\text{total non-failures (FP + TN)}}$$

We answer **RQ 6.4** by running MAGNETO on the output generated by APE [131] and we

Table 6.2: MAGNETO empirical evaluation results.

Oracle Name	Failure ID	Detected	Oracles				
			O1	O2	O3	O4	O5
Orientation change (O1)	o1-1	✓	-	✗	✗	✗	✗
	o1-2	✓	-	✗	✗	✗	✗
	o1-3	✓	-	✗	✗	✗	✗
User-entered data (O2)	o2-1	✓	✗	-	TN	✗	✗
	o2-2	✗	✗	-	✗	✗	✗
	o2-3	✗	✗	-	✗	✗	✗
Back pressed (O3)	o3-1	✓	✗	✗	-	✗	✗
	o3-2	✓	✗	✗	-	✗	✗
	o3-3	✗	✗	TN	-	✗	✗
Language set (O4)	o4-1	✓	✗	✗	FP	-	✗
	o4-2	✗	✗	✗	✗	-	✗
	o4-3	✓	✗	✗	✗	-	✗
Theme set (O5)	o5-1	✓	✗	✗	✗	✗	-
	o5-2	✓	✗	TN	TP	✗	-
	o5-3	✓	✗	✗	✗	✗	-
<b>Total</b>	15	11					

answer **RQ 6.5** by comparing the MAGNETO result with result from OwlEyes [169].

#### 6.4.4 Evaluation Results

First, we provide an overview of the evaluation results of applying MAGNETO to the execution data from our 15 ground truth bugs before answering each RQ in detail. These initial results are shown in Table 6.2. We evaluated in two steps. First, we ran the five oracle types on the 15 failures—three for each oracle type as shown by *Oracle Name* and *Failure ID* in the table. All 15 cases had at least one known failure. MAGNETO revealed 11 of the 15 failures, as shown by the check marks in the *Detected* column (✓). MAGNETO also found a new failure in one of the apps by using the *theme set* oracle.

In the second step, we ran each oracle on the 12 failures that were not related to the same oracle type (*Oracles* in the table), as recall there are 3 failures per oracle type. The five column headers are the oracle names from the first column, where the ‘-’ symbol means

the same oracle.  $\bar{X}$  means the *oracle was not triggered*. TN means *true negative*, that is, the trigger was detected, the oracle was executed, but no failure was detected and none existed. TP means *true positive*, that is, the oracle revealed the failure. FP means *false positive*, that is, the oracle reported a failure when there was not one. Of the 60 executions (12 bugs for five oracles), MAGNETO oracles were correctly *not* triggered 55 times ( $\bar{X}$ ). Of those 60, MAGNETO found five triggers (not related to the bugs) and executed the oracles. There were no failures in three of these cases, and it correctly marked these cases (TN), which we verified by hand. In one case a trigger was detected when it should not have been, resulting in one false positive (FP), for failure ID o4-1. On further analysis, we found that noise in the data extracted from `uiautomator` improperly triggered the oracle. MAGNETO also found an unreported failure in one of the apps. We manually verified the failure and found that the issue had been fixed by developers in a later version of the same app. This failure was found by the *back pressed* oracle for failure ID o5-2.

#### 6.4.4.1 RQ 6.1: How Accurate is MAGNETO’s Oracle Trigger Detection?

To answer RQ 6.1, we analyzed the frequency of MAGNETO’s trigger detection with respect to the actual number of triggers present. MAGNETO’s oracle was correctly triggered 12 times out of 15 executions in step 1, and missed the trigger three times. This results in a trigger detection accuracy of  $\frac{12}{15} = 80\%$ .

For step 2, the oracle was incorrectly triggered once, correctly triggered 4 times, and correctly not triggered 55 times out of 60 executions (12 bugs  $\times$  5 oracles). Thus, MAGNETO’s trigger detection accuracy is  $\frac{4+55}{60} = 98.3\%$ . The overall accuracy for both steps of evaluation is:  $\frac{71}{75} = 94.6\%$ . We examined the two times that a trigger for a bug was missed and the instances where triggers were incorrectly detected. Our current trigger detection technique uses a bag of English keywords along with GUI elements derived from the behavioral oracle taxonomy. If the GUI element used in an app is not considered in our current implementation, the trigger is missed and the oracle is not executed. We also found a case where oracle was triggered when it should not have been, and this was due to an incorrect

Table 6.3: MAGNETO results with APE inputs

Oracle Name	Apps			
	Amaze	Book Catalogue	Flowx	VLC
Orientation change (O1)	∅	∅	∅	∅
User-entered data (O2)	TN	FP	FP	TP
Back pressed (O3)	TN	TN	TN	TN
Language set (O4)	∅	∅	∅	FP
Theme set (O5)	TP	TN	FP	TN

result from `uiautomator`.

#### 6.4.4.2 RQ 6.2: What is MAGNETO’s Success Rate in Detecting Failure?

To answer this question, we analyzed the total number of revealed failures. MAGNETO correctly revealed 11 out of 15 failures (✓ in the *Detected* column). MAGNETO’s failure revealing success rate is  $\frac{11}{15} = 73.3\%$ . The four cases where the oracle did not reveal a failure were due to missed triggers.

#### 6.4.4.3 RQ 6.3: How Reliable is MAGNETO in Signaling Failure?

We answer this research question by analyzing the total number of times MAGNETO incorrectly reported a failure. There were no cases where MAGNETO incorrectly decided there was a failure when executing against the 15 reports. However, when we exposed the oracles to bug reports reporting different expected behavior, we found one false positive. So, the FP rate is  $\frac{1}{1+58} = 1.7\%$ . Further analysis revealed the one FP to be the result of incorrect trigger detection.

#### 6.4.4.4 RQ 6.4: Can MAGNETO be Combined with Existing AIG Tools?

We answer this research question by integrating the APE AIG with trigger detection and oracle execution components of MAGNETO. We executed this integrated setup on four apps

and analyzed their results, shown in Table 6.3. The APE tool did not cover all the five behaviors our oracles check, shown with  $\text{E}$  in Table 6.3. True negatives (TN) means no failure was found or reported. True positive (TP) means the intended action failed and was correctly revealed. MAGNETO found two bugs during the execution of these four apps related to the theme and user-entered data oracles.

During its 30 min execution on the four apps, APE generated 7382 actions, and MAGNETO reported a small number of false positives for the oracles as indicated in Table 6.3. However, given the large number of screens and actions considered, MAGNETO’s false positive rate was extremely small ranging from 0.04% to 2.8% with an average across apps of 0.78%. Analysis showed four reasons for the false positives: (i) insufficient information on user input data (77.59%), (ii) noisy screenshots (1.72%), (iii) unconfirmed app logic (18.97%) like screen scrolling down after input update, and (iv) third party library problems (1.72%).

#### **6.4.4.5 RQ 6.5: How does MAGNETO Compare with Existing Work?**

To answer this question, we ran OwlEyes [169] on the data extracted from the execution of our 15 ground truth bugs. OwlEyes did not reveal any of the failures for these bugs. There is no overlap between the issues that Magneto detects and the issues that OwlEyes detect, signaling that our approach is largely complementary to existing work.

## **6.5 Discussion**

This section discusses the findings from MAGNETO, and some limitations of MAGNETO.

Our behavioral oracle taxonomy is a novel resource that can be used in other research contexts. It identifies and categorizes common GUI element behaviors, and illustrates salient patterns that can be detected and used to automate oracles, as we have demonstrated. We also found that app behavior invariants exist in the wild. Since some invariants are more common than others, prioritizing the more common app behavior invariants for automated

oracle development increases the return on investment of the engineering effort needed in their construction. Our taxonomy is also extensible, and can be used to find more patterns and to derive additional automation approaches. Most importantly, we observed that several App Behavior Invariants exhibited semantically coherent patterns that are ripe for automation, but require more sophisticated program analysis, computer vision, or natural language processing techniques to detect. For example, future work could build techniques that use videos of app executions to identify unexpected behaviors such as when certain types of Dialog boxes should appear.

We found that MAGNETO revealed 11 of 15 failures and others were missed because the oracle was not executed. MAGNETO had few false positives, which means it rarely marks a non-failing app as failing. This is important, because tools that present too many false positives to a developer will be considered unreliable. Execution time of the five oracles ranged from only 0.013s to 5.719s, so the overhead was low enough to allow them to run in real-time with current automated testing techniques. We relied on the taxonomy findings to construct precise oracle triggers that limited false positives and long execution time. Further, MAGNETO revealed two previously unreported failures in two apps (which have been fixed). This demonstrates that the behavior invariants derived from the taxonomy will have broad applicability.

**Limitations.** MAGNETO’s limitations arise from the limited scope of our current oracles, and third party dependencies such as the libraries and tools we use. Similarly, the app behavior invariants are derived from our taxonomy. This poses an external threat to validity. We found that MAGNETO did not reveal failure when the element is not considered a trigger in our current approach and the oracle is not executed. Our current trigger detection technique uses a bag of English keywords along with GUI elements derived from the behavioral oracle taxonomy. In one case, MAGNETO oracles were not executed because the app used German and our trigger detection technique only handles English keywords. However, expanding the current scope of trigger detection technique allowed us to execute the automated oracle on the failing apps and MAGNETO revealed previously missed failures.

MAGNETO does not reveal failures if the expected app behavior was uncommon in our behavior taxonomy and we did not consider that behavior when designing MAGNETO. This limitation can be overcome with additional engineering effort. During our experimentation, we found three cases that pose internal threats to validity:

- We use  $\Delta E \leq 2$  as the threshold for deciding whether themes are equivalent. MAGNETO marked failure when the two colors looked similar by our judgement but had a  $\Delta E$  value of 9.85. Customizing  $\Delta E$  threshold value per app can solve this issue. The threshold value can be set for the app when the developers decide the app color palette and would require minimal effort.
- We used a third party library to detect the language. MAGNETO marked the test to have failed when the library detected the language incorrectly. Our technique can benefit from future advances on language detection.
- We used `uiautomator` to track the components during execution. When `uiautomator` returns an incomplete or incorrect GUI hierarchy in XML, our approach may struggle to properly identify a trigger.

## 6.6 Conclusion and Future Work

This chapter presents results from an in-depth analysis of test oracles for mobile apps. This analysis has led to increased knowledge and understanding of test oracles for mobile apps, and has led to increased ability to create test oracles within automated tests for mobile apps. We identified commonalities (patterns) in faulty and expected behavior of mobile apps, and used them to create two general failure taxonomies for mobile apps. We developed techniques to use these behavior patterns to develop automated test oracles. Existing test approaches for mobile apps rely primarily on crashes, which only account for around 30% of all failures [1, 22]. The key contribution of this chapter is that testers can now create automated test oracles for frequent behaviors, which can allow tests to reveal failures that are not crashes.

Empirical studies on Android apps are expensive, both in terms of resources and time.

Reproducing failures requires significant resources to either reproduce them by hand, or to develop automated failure reproduction scripts. This is partly because failure descriptions do not conform to a standard structure, making it hard to identify clear reproduction steps and failure descriptions, partly because all interactions with mobile apps are through GUIs, and partly because mobile app emulators are quite slow. We also learned that some failing behaviors are more common than others. Automating oracles for these common failures offers a better return on investment.

One challenge to our empirical work is a shortage of automated oracles. The oracles that are available are able to reveal limited failures in limited components. However, our empirical study shows that it is possible to create more test oracles that can cover a broader set of failures and components. We also found a few false positives, where triggers for oracle execution was mistakenly identified. These errors were due to limitations of the tools and libraries we use in MAGNETO.

This chapter makes two major contributions: (1) a pair of mobile app failure taxonomies, one based on how the failure affects users and the other based on how the failure manifests through the GUI, (2) a procedure to develop automated test oracles for GUI-based testing of mobile apps. The behavioral oracle taxonomies add significant understanding to the failure and expected behavior patterns. We used these taxonomies to derive five app-independent automated oracle implementations that verifies five app behavior invariants. We empirically evaluated our oracle implementations and report on their robustness and generalizability.

We hope that the behavior patterns in the behavioral taxonomies can be used to further analyze the app behaviors and lead to additional test oracle research for mobile apps and other GUI-based software. Further, the empirical results can guide researchers to increase our ability to address the test oracle problem. The successful test oracle implementations demonstrate the applicability and potential effectiveness of automation for GUI testing. This research also shows the importance of well-described bug reports. This observation adds to the argument for developing standard structure and practice for bug reports. Lastly, the effectiveness of the test oracles we developed demonstrates the potential for practical



use. Our results can be embedded in test frameworks and even programming language designs to be used by practicing mobile app developers and testers to find and eliminate software and GUI faults before they are inflicted on users.

Future work can generalize this approach with additional types of test oracles, evaluate on more mobile apps and more failures, and improve the research demonstration software to reveal additional failures. Furthermore, researchers can explore new methods of detecting and understanding GUI-based behavior in mobile applications, based primarily on advancements in neural computer vision techniques, to automate both test input generation and fault detection.

## Chapter 7: Conclusion and Future Research

This chapter revisits the problem statement, the thesis statement, research questions, and draws conclusion in Section 7.1. The chapter then lists the contribution of this dissertation in Section 7.2. Section 7.3 discusses the impact of the dissertation and Section 7.4 lists the paper I contributed to during my dissertation. Finally, the chapter concludes with future research directions in Section 7.5.

### 7.1 Research Conclusion

As software becomes more and more entwined in our lives, concerns of software security, cost, efficiency has increased the need and interest in software quality. There can be flaws in software design, or developments that compromise software quality. Software testing is done to find such flaws, and ensure that the software performs in a safe, expected manner without any faults. However, if such flaws go unidentified, they can lead to situations worse than minor bugs, or user inconvenience. There are numerous cases where low quality software resulted in loss of millions of dollars and sometimes even lives. Thus, we need approaches to effectively test software to ensure sufficient software quality.

This research focused on test oracles and test maintenance. The problem statement of this dissertation from Section 1.3 was:

Problem Statement:

Incorrect test oracles, and test bloat reduces test suite quality, which leads to low quality software.

The thesis statement of this dissertation from Section 1.4 was:

Thesis Statement:

Effective automated test oracles and automated test maintenance can improve test suite quality, which in turn will lead to higher quality software.

This thesis statement has been validated with seven research questions across three studies.

The first study (chapter 4) investigated blind tests. A blind test has an incorrect test oracle where a key part of the output is not observed by the oracle, making the test blind. This study investigates the pervasiveness of blind tests as well as its root cause through empirical study of tests written by students and professionals.

- RQ 4.1 What percentage of tests are blind?
  - In the first study with students, on average, 70.38% of tests were blind.
  - In the second study with students, on average, 38.97% of tests were blind.
  - In the third study with professional, on average, 95% of tests were blind.
- RQ 4.2 Do some groups of testers write more blind tests than other groups?
  - We found that students in the both the studies performed better than the professionals. In the study with professionals, 95% of the failure causing tests they wrote were blind.

This study also sheds light on some root causes of blind tests such as code reuse, misunderstood program requirements, technical inexperience, and lack of testing knowledge. Although this list of root causes is not complete, it gives us a strong starting point for further research.

The next study (chapter 5) focused on designing a framework to tackle one of the causes of incorrect test oracles—misunderstood requirements and difficulties in test management. This study developed a framework to automate test management by making tests smarter. A test is *smart* if it can decide when it needs to be run, ignored, modified, or discarded after the software is changed, with few to no developer interaction. This research includes an

empirical study to evaluate the accuracy of framework suggested actions and its execution cost with two research questions:

- RQ 5.1 How much time does the analysis use?
  - The framework took 6.87 seconds on average to analyze each mutation.
  - Further work on code optimization, reduction of unnecessary input output operations can be used to lower the time.
- RQ 5.2 How accurate are the analysis results from the framework?
  - The smart test framework accuracy was found to be 94%.
  - Further analysis of the incorrect results from the framework showed that improving program representation like control flow graph can increase the framework accuracy.

After evaluating the applicability of the test management framework, the next study (chapter 6) investigated test oracle automation approach for mobile apps. The study modeled common GUI app behaviors to derive a novel *behavioral oracle taxonomy* and developed an automated approach MAGNETO (autoMATinG aNdroid tEsT Oracles) to automatically detect failures. The study then empirically evaluated the five automated oracle patterns on 15 real-world faults mined from open source Android apps with following research questions:

- RQ 6.1 How accurate is MAGNETO’s oracle trigger detection?
  - We evaluate MAGNETO’s oracle trigger detection accuracy in two steps: (a) failure causing trigger accuracy, and (b) benign trigger accuracy
  - Failure causing trigger accuracy shows MAGNETO’s trigger accuracy when run on all oracle triggering scenarios. MAGNETO’s trigger detection accuracy is 80%. It detected 12 of the 15 failure causing triggers correctly and missed the remaining.
  - Benign trigger accuracy shows MAGNETO’s trigger accuracy when run on triggering and non-triggering scenarios. MAGNETO’s trigger detection accuracy is 98.3%. It correctly triggered the oracle 4 times and correctly skipped the oracle execution 55 times out of 60 executions.
  - The overall accuracy is 94.6%.
- RQ 6.2 What is MAGNETO’s success rate in detecting failure?

- MAGNETO correctly revealed 11 out of 15 failures. The failure revealing success rate is 73.3%.
- RQ 6.3 How reliable is MAGNETO in signaling failures?
  - MAGNETO had no false positive result when run on scenarios resulting in failures.
  - MAGNETO had one false positive when run on failing as well as failure-free scenarios.
- RQ 6.4 Can MAGNETO be combined with existing AIG tools?
  - MAGNETO integrated with APE AIG tool found two bugs related to theme and user-entered data.
  - MAGNETO’s false positive rate was 0.78% on average when run on APE generated inputs.
- RQ 6.5 How does MAGNETO compare with existing work that detects UI Display Issues?
  - Our approach is complementary to existing work. We found no overlap between the issues that MAGNETO detects and the issues that a recent work, OwlEyes detect [169]. Further, OwlEyes did not reveal any of the failure when run on our 15 ground truth bugs.

## 7.2 Contributions Summary

This research project increased our understanding of two key aspects of test automation. We developed strategies, techniques, algorithms, and tools to automatically create automated test oracles that are more effective than existing test oracles. We also invented techniques, algorithms, and tools to effectively automate test maintenance activities that are currently done by hand.

The ultimate goal of this research is to improve software quality, through more effective and efficient testing. This research makes five major contributions:

1. A study of blind tests: Awareness is the first step to action. With this empirical study, the prevalence of blind tests among automated tests written by professional and non-professionals has been brought to light along with its root causes. The study shows that blind tests are a major problem for test automation.

2. A test maintenance framework: I developed a test maintenance framework to overcome the issue of test bloat. The framework maps test requirements to automated tests and based on this mapping analyzes if there are unnecessary tests causing test bloats. The framework also recommends appropriate action when source code changes. This framework is compatible with Java applications and JUnit tests.
3. GUI-based behavioral oracle taxonomy: This research delivers a taxonomy of mobile app behaviors derived by analyzing and categorizing bug reports. This taxonomy can be further analyzed and built upon by developers and researchers to identify more app invariants and build more automated oracles.
4. Automated test oracles based on the behavioral oracle taxonomy: This research identified recurring app behaviors as app behavior invariants from the taxonomy. I then developed automated test oracles to detect the app behavior invariants.
5. Evaluation of automated test oracles: This research evaluated the success rate and reliability of automated test oracles. It also evaluated compatibility of automated test oracles with existing automated input generation tools.

### **7.3 Impact**

This dissertation has impacts for both professional software engineers and for the research community.

For professional software engineers, test maintenance and the test oracle problem are well known challenges. This research presents new ideas to partially solve the test oracle problem and the test maintenance problem. The results from the study on blind tests can be used to educate developers and testers, and to create educational materials on how to avoid creating blind tests. Similarly, the underlying ideas of the smart test framework, such as self-awareness, and self-determination, can be applied on a larger scale to address the test maintenance problem. With GUI testing becoming more and more prevalent in industry, techniques that enable test oracle automation is beneficial. The insights from the behavioral oracle taxonomy can also guide developers in deciding which category of failures

are more suitable for automation and which can be prioritized for manual testing.

Researchers can use the study on blind tests as a starting point and further explore its cause, and seek additional ways to mitigate it. The idea of self-aware, self-managing smart tests is a novel approach to test suite management and has potential for further exploration and scope expansion. Finally, this research presents the first GUI-based behavioral oracle taxonomy for mobile apps that describes the characteristic patterns of different types of failures that have occurred in Android apps “in the wild.” This taxonomy is extensible and can be used to find more patterns and derive more automation approaches.

## 7.4 Papers

This section lists the paper that are based on this dissertation as well as additional papers that I have co-authored during my Ph.D. study.

### 1. Papers based on this dissertation:

- (a) Kesina Baral, and Jeff Offutt. An Empirical Analysis of Blind Tests. IEEE International Conference on Software Testing, Validation and Verification (ICST), Portugal, September 2020 [22].
- (b) Kesina Baral, Jeff Offutt, and Fiza Mulla. Self determination: A comprehensive strategy for making automated tests more effective and efficient. IEEE Conference on Software Testing, Verification and Validation (ICST), Brazil, April 2021 [23].
- (c) Kesina Baral, Jack Johnson, Junayed Mahmud, Sabiha Salma, Mattia Fazzini, Jeff Offutt, Julia Rubin, and Kevin Moran. MAGNETO: Automating GUI-based Test Oracles for Mobile Apps. Manuscript under review.

### 2. Other papers I have co-authored during my Ph.D.:

- (a) Kesina Baral, Rasika Mohod, Jennifer Flamm, Seth Goldrich, and Paul Ammann. Evaluating a test automation decision support tool. IEEE International Conference on Software Testing, Verification and Validation, China, April 2019 [172].

- (b) Jeff Offutt, Birgitta Lindström, and Kesina Baral. Teaching an International Distributed Discussion-Based Course. Foundations on Education in Computer Science, Las Vegas, July 2019 [173].
- (c) Kesina Baral, and Paul Ammann. Teaching a Testing Concept (JUnit) with Active Learning. IEEE International Conference on Software Testing, Verification and Validation Workshops -TestEd, Portugal, September 2020 [174].
- (d) Marcos Lordello Chaim, Kesina Baral, Jeff Offutt, Mario Concilio Neto, and Roberto Araujo. Efficiently Finding Data Flow Subsumptions. IEEE Conference on Software Testing, Verification and Validation (ICST), Brazil, April 2021 [175].
- (e) Kesina Baral, Jeff Offutt, Paul Ammann, and Rasika Mohod. Practice Makes Better: Quiz Retake Software to Increase Student Learning. Education through Advanced Software Engineering and Artificial Intelligence (EASEAI), Co-located with ESEC/FSE '21, Greece, August 2021 [176].
- (f) Yixue Zhao, Saghar Talebipour, Kesina Baral, Hyojae Park, Leon Yee, Safwat Ali Khan, Yuriy Brun, Nenad Medvidovic, and Kevin Moran. AVGUST: Automating Usage-Based Test Generation from Videos of App Executions. Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22) [177].
- (g) Jeff Offutt, and Kesina Baral. Designing Divergent Thinking, Creative Problem Solving Exams. Software Engineering Education and Training (SEET) track at the International Conference on Software Engineering (ICSE '22), USA, May 2022 [178].

## 7.5 Future Research

This research designed, implemented, and evaluated approaches and tools to improve test quality by tackling challenges of test oracle problem and test maintenance. Although the research takes significant steps towards solving the problems, there are more avenues for improvement and future research. This section proposed further research direction:



1. Test Oracle Problem: The research presented in chapter 4 identifies blind tests as a common but not widely studied test oracle problem. This problem can be addressed in multiple ways: (a) educating software engineers about this issue, and providing training on how to write effective test oracles; (b) developing approaches to detect issues in test oracles; and (c) developing technique to repair the issues discovered in test oracles through automatic program repair techniques.

The research described in chapter 6 presents a pair of taxonomies that depict the failure effect on users and failure manifestation through GUIs. It also contributes automated test oracles based on these taxonomies. The taxonomy can be used to further analyze the app behaviors and lead to additional test oracle research. This research also affirms the importance of descriptive bug reports and the need for a standard practice. In future, the taxonomy can be expanded and used to identify more behavior invariants. Furthermore, additional oracles can be automated and embedded into test frameworks to make it easier for developers to find GUI faults.

2. Test Maintenance: The research presented in chapter 5 presents a strategy to tackle test maintenance problem. Currently, the test maintenance framework alerts developers for manual intervention if tests need to be added or edited. One future direction for this research is to use automatic repair techniques to update the tests as required. Similarly, the framework is currently limited to detecting changes in one line, but can be expanded to detect larger changes. Additional evaluation on open source software will also help in investigating and identifying effective approaches for test maintenance problems.

# Appendix A: Appendix

### Quiz Retake Scheduler

You can sign up for quiz retakes within the next two weeks.  
Today is *WEDNESDAY, FEBRUARY 10*. Currently scheduling quizzes for the next two weeks, until *WEDNESDAY, FEBRUARY 24*.  
Enter your name (as it appears on the class roster), then select which date, time, and quiz you wish to retake from the following list.

Name:

Friday, February 12 , at 15:00 in CS Building 1000	<input type="checkbox"/>
Quiz 2 from Friday, February 5	<input type="checkbox"/>
Quiz 3 from Friday, February 12	<input type="checkbox"/>
Saturday, February 13 , at 15:30 in CS Building 1001	<input type="checkbox"/>
Quiz 2 from Friday, February 5	<input type="checkbox"/>
Quiz 3 from Friday, February 12	<input type="checkbox"/>
Sunday, February 14 , at 10:00 in Engineering Hall	<input type="checkbox"/>
Quiz 2 from Friday, February 5	<input type="checkbox"/>
Quiz 3 from Friday, February 12	<input type="checkbox"/>
Friday, February 19 , at 15:00 in CS Building 1000	<input type="checkbox"/>
Quiz 3 from Friday, February 12	<input type="checkbox"/>
Quiz 4 from Friday, February 19	<input type="checkbox"/>
Saturday, February 20 , at 15:30 in CS Building 1001	<input type="checkbox"/>
Quiz 3 from Friday, February 12	<input type="checkbox"/>
Quiz 4 from Friday, February 19	<input type="checkbox"/>
Sunday, February 21 , at 10:00 in Engineering Hall	<input type="checkbox"/>
Quiz 3 from Friday, February 12	<input type="checkbox"/>
Quiz 4 from Friday, February 19	<input type="checkbox"/>
<input type="button" value="Submit request"/>	

Figure A.1: Quiz retake scheduler web version

## 1.1 Individual Contributions to the Blind Tests Project

- Kesina Baral: Kesina was the lead researcher who conducted the empirical study. She collected data, developed baseline result for the collected data, and analyzed the data. Kesina also evaluated the study result and wrote the paper.
- Jeff Offutt: Jeff served as the faculty advisor on this project and helped to guide the design and evaluation of the study. He was also involved in collecting the data from student subjects. Jeff helped in revising the paper for peer-review.

## 1.2 Individual Contributions to the Smart Tests Project

- Kesina Baral: Kesina was the lead researcher who drove the conceptualization and development of the smart test framework for test suite maintenance. She designed the framework architecture and implemented all the components of the framework. Kesina also formulated the evaluation strategy, conducted the framework evaluation, and wrote the paper.
- Jeff Offutt: Jeff served as the faculty advisor on this project and helped to guide the design and evaluation of the study. Jeff was heavily involved in revising the paper for peer-review.
- Fiza Mulla: Fiza worked on setting up open source projects for the framework evaluation.

## 1.3 Individual Contributions to the MAGNETO Project

- Kesina Baral: Kesina was the lead researcher who drove the conceptualization and development of the behavioral oracle taxonomy and MAGNETO's approach. She also lead the qualitative analysis of bug reports that led to the development of behavioral oracle taxonomy and identification of the behavior invariants. Kesina designed the overall architecture of MAGNETO's approach and implemented all the components of the approach. She formulated the evaluation strategy, conducted the evaluation, and wrote the paper.

- Jeff Offutt: Jeff served as the faculty advisor on this project and helped to guide the design and evaluation of the project. Jeff was heavily involved in revising the paper for peer-review.
- Kevin Moran: Kevin aided in designing the qualitative analysis approach for behavioral oracle taxonomy development. He also helped to guide MAGNETO's implementation and evaluation strategy. Kevin was heavily involved in the manuscript preparation.
- Jack Johnson: Jack primarily aided in qualitative analysis of bug reports and evaluation data collection.
- Junayed Mahmud: Junayed primarily worked on qualitative analysis of bug reports for taxonomy development.
- Sabiha Salma: Sabiha primarily worked on qualitative analysis of bug reports for taxonomy development.
- Mattia Fazzini: Mattia aided in designing the evaluation strategy and manuscript preparation.
- Julia Rubin: Julia aided in designing the evaluation strategy and manuscript preparation.

Listing A.1: Quiz schedule code

---

```
public static void printQuizScheduleForm(quizzes quizList, retakes retakesList,
    courseBean course) {
    // Check for a week to skip
    boolean skip = false;
    LocalDate startSkip = course.getStartSkip();
    LocalDate endSkip = course.getEndSkip();

    System.out.println("");
    System.out.println("");
    System.out.println("*****");
    System.out.println("University quiz retake scheduler for class " +
        course.getCourseTitle());
    System.out.println("*****");
```

```

System.out.println("");
System.out.println("");

// print the main form
System.out.println("You can sign up for quiz retakes within the next two weeks.
    ");
System.out.println("Enter your name (as it appears on the class roster), ");
System.out.println("then select which date, time, and quiz you wish to retake
    from the following list.");
System.out.println("");

LocalDate today = LocalDate.now();
LocalDate endDay = today.plusDays(new Long(daysAvailable));
LocalDate origEndDay = endDay;
/*
 * Faulty * fault 1, should check if endDay and today is between
 * startSkip and endSkip
 */

if (!endDay.isBefore(startSkip) && !endDay.isAfter(endSkip)) {
    endDay = endDay.plusDays(new Long(7));
    skip = true;
}

System.out.print("Today is ");
System.out.println((today.getDayOfWeek()) + ", " + today.getMonth() + " " +
    today.getDayOfMonth());
System.out.print("Currently scheduling quizzes for the next two weeks, until ");
System.out.println((endDay.getDayOfWeek()) + ", " + endDay.getMonth() + " " +
    endDay.getDayOfMonth());
System.out.print("");
int quizRetakeCount = 0;

```

```

for (retakeBean r : retakesList) {
    LocalDate retakeDay = r.getDate();
    /* FAULTY * fault 5, only checks for two week duration, shows retake
    * during skip week too
    */

    if (!(retakeDay.isBefore(today)) && !(retakeDay.isAfter(endDay))) {
        /* FAULTY * fault 4, prints skip message for retake after 2 week
        * period instead of the first retake after skip week.
        */
        if (skip && retakeDay.isAfter(origEndDay)) {
            System.out.println("Skipping a week, no quiz or retakes.");
            skip = false;
        }
        // format: Friday, January 12, at 10:00am in EB 4430
        System.out.println("RETAKE: " + retakeDay.getDayOfWeek() + ", " +
            retakeDay.getMonth() + " "
            + retakeDay.getDayOfMonth() + ", at " + r.timeAsString() + " in " +
            r.getLocation());

        for (quizBean q : quizList) {
            LocalDate quizDay = q.getDate();
            LocalDate lastAvailableDay = quizDay.plusDays(new Long(daysAvailable));

            /* To retake a quiz on a given retake day, the retake day must be
            * within two ranges:
            * quizDay <= retakeDay <= lastAvailableDay and
            * today <= retakeDay <= endDay
            */

            /* FAULTY * fault 2, does not increase lastAvailableDay by 7 */
            /* FAULTY * fault 3, only compares day and not quiz time */

```

```

if (!quizDay.isAfter(retakeDay) && !retakeDay.isAfter(lastAvailableDay) &&
    !today.isAfter(retakeDay) && !retakeDay.isAfter(endDay)) {
    quizRetakeCount++;
    retakeQuizIDProps.setProperty(String.valueOf(quizRetakeCount),
    r.getID() + separator + q.getID());
    System.out.print(" " + quizRetakeCount + " ");
    System.out.println("Quiz " + q.getID() + " from " + quizDay.getDayOfWeek()
        + ", "
        + quizDay.getMonth() + " " + quizDay.getDayOfMonth());
    }
}
}
}
}
System.out.println("");
}

```

---

Listing A.1: Quiz schedule code

Listing A.2: Calendar code

---

```

/*****
*
* This program reads the month and year from the command line
* and prints a calendar for that month.
*
* Enter month: 7
* Enter year: 2015
* July 2015
* S M T W Th F S
*           1  2
*  3  4  5  6  7  8  9
* 10 11 12 13 14 15 16
* 17 18 19 20 21 22 23

```

```

* 24 25 26 27 28 29 30
* 31
*
*****/

/* FAULTY * This is a faulty version. */
/* FAULTY * the string FAULTY documents the bugs. */
public class Calendar
{
    /*****
    * Given the month, day, and year, return which day
    * of the week it falls on according to the Gregorian calendar.
    * For month, use 1 for January, 2 for February, and so forth.
    * Returns 0 for Sunday, 1 for Monday, and so forth.
    *****/

    private static int weekDay(int month, int day, int year) {
        /* FAULTY * fault 4, should divide by 12, not 10 */
        int y = year - (14 - month) / 10;
        int x = y + y/4 - y/100 + y/400;
        int m = month + 12 * ((14 - month) / 12) - 2;
        int wd = (day + x + (31*m)/12) % 7;
        return wd;
    }

    private static boolean isLeapYear(int year) {
        /* FAULTY * fault 2, should be "&&", not "||" */
        if ((year % 4 == 0) || (year % 100 != 0))
            return true;
        if (year % 400 == 0)
            return true;
        return false;
    }
}

```



```

}

private static String readInput(Scanner sc, String arg) {
    System.out.print("Enter "+arg+": ");
    return(sc.next());
}

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int month = Integer.parseInt(readInput(sc, "month"));
    int year = Integer.parseInt(readInput(sc, "year"));
    String[] months = {
        "",
        "January", "February", "March",
        "April", "May", "June",
        "July", "August", "September",
        "October", "November", "December"
    };

    /* FAULTY * fault 3, August should be 31, not 30 */
    int[] days = { 0, 31, 28, 31, 30, 31, 30, 31, 30, 30, 31, 30, 31 };

    if (month == 2 && isLeapYear(year))
        /* FAULTY * fault 5, should divide be 29, not 30 */
        days[month] = 30;

    System.out.println(" " + months[month] + " " + year);
    System.out.println(" S M Tu W Th F S");

    int wd = weekDay(month, 1, year);
    for (int i = 0; i < wd; i++)
        System.out.print(" ");

```

```

for (int i = 1; i <= days[month]; i++) {
    System.out.printf("%2d ", i);
    if (((i + wd) % 7 == 0) || (i == days[month]))
        System.out.println();
    }
}
}

```

---

Listing A.2: Calendar code

Table A.1: GUI issues and GUI sub-issues categories

GUI Issues Categorization	GUI Sub-Issues Categorization
Layout	component size component location
Text	font color font style incorrect text content incorrect language
Resource	missing component additional component image color incorrect image component type component shape component theme(font, size, color) component property(required, enabled, selected, checked)
Functionality	incorrect functionality

Table A.2: Complete taxonomy result

Oracle label	GUI Issue label	GUI Sub-Issue label	Affected components	
Text analysis [92]	Incorrect functionality [21]		Screen [1]	
			Bottom navigation bar [2]	
			Button[1]	
			File [2]	
			Image button [1]	
			List [1]	
			Scroll view [1]	
			Image [1]	
			Dialog box [2]	
			Text element [5]	
			Toast [1]	
			Editable text [2]	
			List item [1]	
	Layout [4]	Component location [3]		List [1]
				List item [1]
				Button [1]
	Resource [14]	Component state [2]		Button [1]
				Editable text [1]
		Extraneous component [6]		List item [1]
				Text element [3]

Continued on next page

Table A.2: Complete taxonomy result (Continued)

Oracle label	GUI Issue label	GUI Sub-Issue label	Affected components
			Dialog box [1]
			Toast [1]
		Missing component [6]	Text element [4]
			List [1]
			Image [1]
		Text [53]	Extraneous text [4]
	Toast [2]		
	Incorrect text [35]		Text element [20]
			Spinner [3]
			Editable text [8]
			Tool tip [1]
			Dialog box [2]
			List item [1]
	Missing text [10]		Text element [7]
			Tabbed view [1]
			Editable text [8]
			Checkbox text [1]
	Incorrect language [3]		Text element [1]
	Missing component [1]	Tabbed view [1]	
	Visibility analysis [70]	Incorrect functionality [14]	
Toolbar [1]			

Continued on next page

Table A.2: Complete taxonomy result (Continued)

Oracle label	GUI Issue label	GUI Sub-Issue label	Affected components	
			Bottom navigation bar [1]	
			File [1]	
			Keyboard [1]	
			Image [1]	
			Text element [1]	
			List item [1]	
			Dialog box [1]	
	Layout [6]	Component location [2]		List [1]
				Button [1]
		Component size [4]		Context menu [1]
				List item [1]
				Listview [1]
			Component theme [1]	Screen [1]
			Extraneous component [9]	Keyboard [1]
				Spinner [2]
				Item in list [2]
				Image button [2]
				Image [2]
	Dialog box [2]			

Continued on next page

Table A.2: Complete taxonomy result (Continued)

Oracle label	GUI Issue label	GUI Sub-Issue label	Affected components		
			List [2]		
		Missing component [26]	Dialog box [5]		
			Image [4]		
			Text element [5]		
			List [1]		
			Image [1]		
			Tabbed view [1]		
			List item [3]		
			Editable text [1]		
			Option menu [1]		
			Slide view [1]		
			Button [1]		
			Extraneous text [3]	Text element [2]	
		Toast [1]			
		Incorrect text [2]	Text element [2]		
		Missing text [3]	Editable text [1]		
			Keyboard [1]		
			Tabbed view [1]		
		Image analysis [48]	Incorrect functionality [15]		Screen [2]
				List [1]	
Image [2]					

Continued on next page

Table A.2: Complete taxonomy result (Continued)

Oracle label	GUI Issue label	GUI Sub-Issue label	Affected components		
			Scroll view [1]		
			Text element [2]		
			Toggle button [1]		
			File [1]		
			Toast [1]		
			Image button [1]		
			Editable text [1]		
			Seek bar [1]		
			Dialog box [1]		
	Layout [1]	Component location [1]	List [1]		
	Resource [23]	Component state [6]		Editable text [3]	
				Spinner [1]	
				Image [1]	
				Keyboard [1]	
				Keyboard [1]	
				Text element [1]	
		Extraneous component [3]			List item [1]
					Dialog box [1]
					Image [3]
Missing component [7]			Editable text [1]		

Continued on next page

Table A.2: Complete taxonomy result (Continued)

Oracle label	GUI Issue label	GUI Sub-Issue label	Affected components
			List [1]
			Keyboard [1]
		Image color [2]	Image [2]
		Incorrect image [3]	Image [2]
			List item [1]
		Incorrect webview [1]	Webview [1]
	Text [9]	Extraneous text [2]	Text element [1]
			Toast [1]
		Incorrect text [2]	Text element [1]
		Missing text [4]	Editable text [2]
			Text element [2]
	Incorrect language [1]	Text element [1]	
	Color analysis [8]	Resource[8]	Component theme [8]
Screen [1]			
Toolbar [1]			
Radio button [1]			
Text element [1]			
Checkbox text [1]			
Position analysis [5]	Layout[5]	Component location [5]	List item [1]
			Image button [3]
			Menu [1]

Continued on next page



Table A.2: Complete taxonomy result (Continued)

Oracle label	GUI Issue label	GUI Sub-Issue label	Affected components
			Text element [1]
Webview analysis [8]	Incorrect functionality [3]		Bottom navigation bar [2]
			Image button [1]
State analysis [6]	Incorrect functionality [1]		Toggle button [1]
			Resource [8]
	Image button [1]		
	Spinner [1]		
	Component state selected [2]	Radio button [1]	
Checkbox [1]			
Size analysis [5]	Layout [5]	Component size [4]	Text element [1]
			Menu [1]
			List [1]
			List item [1]
		Text padding [1]	List item [1]
Data analysis [48]	Resource [2]	Additional component [2]	Screen [1]
	Text [1]	Additional text [1]	Editable text [1]
Theme analysis [48]	Resource [2]	Component theme [2]	Toolbar [1]
			Editable text [1]

Continued on next page

Table A.2: Complete taxonomy result (Continued)

Oracle label	GUI Issue label	GUI Sub-Issue label	Affected components
	Text[1]	Font style [1]	Text element [1]

Listing A.3: Example JSON format for MAGNETO execution

```

{
  "date": "Feb 28, 2022, 1:03:43 PM",
  "deviceDimensions": "1080x1920",
  "executionType": "User-Trace-",
  "executionNum": 12,
  "crash": false,
  "deviceName": "Android SDK built for x86",
  "elapsedTime": 841939,
  "orientation": 0,
  "mainActivity": "com.ichi2.anki.DeckPicker",
  "androidVersion": "8.1.0",
  "steps": [{
    "action": 22,
    "sequenceStep": 1,
    "screenshot":
      "com.ichi2.anki.User-Trace.12.com.ichi2.anki_97_anki1_augmented.png",
    "textEntry": "adb -s emulator-5554 shell input
      touchscreen swipe 314 546 282 1421 919",
    "areaEdit": 0,
    "hashStep": "1643",
    "AreaView": 0,
    "areaList": 0,
    "areaSelect": 0,
    "initialX": 0,
    "initialY": 0,
    "finalX": 0,
    "finalY": 0,
    "useCaseTranType": 0,
    "network": false,
    "temperature": false,
    "gps": false,
    "screen": {
      "activity": "Info(Window\u003dMain)",
      "window": "ACTIVITY:com.android.launcher2.Launcher",
      "dynGuiComponents": [{
        "activity": "com.ichi2.anki.Info",
        "name": "",
        "idXml": "NO_ID",
        "componentIndex": 0,
        "componentTotalIndex": 0,
        "positionX": 0,
        "positionY": 0,
        "height": 0,
        "width": 0,

```

```

    "checkable": false,
    "checked": false,
    "clickable": false,
    "enabled": false,
    "focusable": false,
    "focused": false,
    "longClickable": false,
    "scrollable": false,
    "selected": false,
    "password": false,
    "itemList": false,
    "calendarWindow": false,
    "relativeLocation": "Top left",
    "idText": "NO_ID",
    "offset": 0,
    "drawTime": 0.0
}, {
    "activity": "com.ichi2.anki.Info",
    "name": "android.widget.FrameLayout",
    "text": "",
    "contentDescription": "",
    "componentIndex": 0,
    "componentTotalIndex": 0,
    "positionX": 0,
    "positionY": 0,
    "height": 1794,
    "width": 1080,
    "checkable": false,
    "checked": false,
    "clickable": false,
    "enabled": true,
    "focusable": false,
    "focused": false,
    "longClickable": false,
    "scrollable": false,
    "selected": false,
    "password": false,
    "itemList": false,
    "calendarWindow": false,
    "relativeLocation": "Center",
    "idText": "",
    "offset": 0,
    "drawTime": 0.0
}, {
    "activity": "com.ichi2.anki.Info",
    "name": "android.view.ViewGroup",
    "text": "",
    "contentDescription": "",
    "idXml": "android:id/decor_content_parent",
    "componentIndex": 0,
    "componentTotalIndex": 0,
    "positionX": 0,
    "positionY": 0,
    "height": 1794,
    "width": 1080,
    "checkable": false,
    "checked": false,

```

```

    "clickable": false,
    "enabled": true,
    "focusable": false,
    "focused": false,
    "longClickable": false,
    "scrollable": false,
    "selected": false,
    "password": false,
    "itemList": false,
    "calendarWindow": false,
    "relativeLocation": "Center",
    "idText": "android:id/decor_content_parent",
    "offset": 0,
    "drawTime": 0.0
  }, {
    "activity": "com.ichi2.anki.Info",
    "name": "android.widget.FrameLayout",
    "text": "",
    "contentDescription": "",
    "idXml": "android:id/action_bar_container",
    "componentIndex": 0,
    "componentTotalIndex": 0,
    "positionX": 0,
    "positionY": 63,
    "height": 126,
    "width": 1080,
    "checkable": false,
    "checked": false,
    "clickable": false,
    "enabled": true,
    "focusable": false,
    "focused": false,
    "longClickable": false,
    "scrollable": false,
    "selected": false,
    "password": false,
    "itemList": false,
    "calendarWindow": false,
    "relativeLocation": "Top",
    "idText": "android:id/action_bar_container",
    "offset": 0,
    "drawTime": 0.0
  }, {
    "activity": "com.ichi2.anki.Info",
    "name": "android.view.ViewGroup",
    "text": "",
    "contentDescription": "",
    "idXml": "android:id/action_bar",
    "componentIndex": 0,
    "componentTotalIndex": 0,
    "positionX": 0,
    "positionY": 63,
    "height": 126,
    "width": 1080,
    "checkable": false,
    "checked": false,
    "clickable": false,

```

```

    "enabled": true,
    "focusable": false,
    "focused": false,
    "longClickable": false,
    "scrollable": false,
    "selected": false,
    "password": false,
    "itemList": false,
    "calendarWindow": false,
    "relativeLocation": "Top",
    "idText": "android:id/action_bar",
    "offset": 0,
    "drawTime": 0.0
}, {
    "activity": "com.ichi2.anki.Info",
    "name": "android.widget.LinearLayout",
    "text": "",
    "contentDescription": "",
    "idXml": "",
    "componentIndex": 0,
    "componentTotalIndex": 0,
    "positionX": 21,
    "positionY": 63,
    "height": 126,
    "width": 570,
    "checkable": false,
    "checked": false,
    "clickable": true,
    "enabled": false,
    "focusable": false,
    "focused": false,
    "longClickable": false,
    "scrollable": false,
    "selected": false,
    "password": false,
    "itemList": false,
    "calendarWindow": false,
    "relativeLocation": "Top left",
    "idText": "",
    "offset": 0,
    "drawTime": 0.0
}, {
    "activity": "com.ichi2.anki.Info",
    "name": "android.widget.FrameLayout",
    "text": "",
    "contentDescription": "",
    "idXml": "",
    "componentIndex": 0,
    "componentTotalIndex": 0,
    "positionX": 21,
    "positionY": 63,
    "height": 126,
    "width": 105,
    "checkable": false,
    "checked": false,
    "clickable": false,
    "enabled": true,

```

```

    "focusable": false,
    "focused": false,
    "longClickable": false,
    "scrollable": false,
    "selected": false,
    "password": false,
    "itemList": false,
    "calendarWindow": false,
    "relativeLocation": "Top left",
    "idText": "",
    "offset": 0,
    "drawTime": 0.0
  }, {
    "activity": "com.ichi2.anki.Info",
    "name": "android.widget.ImageView",
    "text": "",
    "contentDescription": "",
    "idXml": "android:id/home",
    "componentIndex": 0,
    "componentTotalIndex": 0,
    "positionX": 31,
    "positionY": 84,
    "height": 84,
    "width": 84,
    "checkable": false,
    "checked": false,
    "clickable": false,
    "enabled": true,
    "focusable": false,
    "focused": false,
    "longClickable": false,
    "scrollable": false,
    "selected": false,
    "password": false,
    "itemList": false,
    "calendarWindow": false,
    "relativeLocation": "Top left",
    "idText": "android:id/home",
    "offset": 0,
    "drawTime": 0.0
  }, {
    "activity": "com.ichi2.anki.Info",
    "name": "android.widget.LinearLayout",
    "text": "",
    "contentDescription": "",
    "idXml": "",
    "componentIndex": 1,
    "componentTotalIndex": 0,
    "positionX": 126,
    "positionY": 94,
    "height": 63,
    "width": 465,
    "checkable": false,
    "checked": false,
    "clickable": false,
    "enabled": true,
    "focusable": false,

```

```

    "focused": false,
    "longClickable": false,
    "scrollable": false,
    "selected": false,
    "password": false,
    "itemList": false,
    "calendarWindow": false,
    "relativeLocation": "Top left",
    "idText": "",
    "offset": 0,
    "drawTime": 0.0
  }, {
    "activity": "com.ichi2.anki.Info",
    "name": "android.widget.TextView",
    "text": "AnkiDroid v2.3alpha3",
    "contentDescription": "",
    "idXml": "android:id/action_bar_title",
    "componentIndex": 0,
    "componentTotalIndex": 0,
    "positionX": 126,
    "positionY": 94,
    "height": 63,
    "width": 444,
    "checkable": false,
    "checked": false,
    "clickable": false,
    "enabled": true,
    "focusable": false,
    "focused": false,
    "longClickable": false,
    "scrollable": false,
    "selected": false,
    "password": false,
    "itemList": false,
    "calendarWindow": false,
    "relativeLocation": "Top left",
    "idText": "android:id/action_bar_title",
    "offset": 0,
    "drawTime": 0.0
  }, {
    "activity": "com.ichi2.anki.Info",
    "name": "android.widget.FrameLayout",
    "text": "",
    "contentDescription": "",
    "idXml": "android:id/content",
    "componentIndex": 1,
    "componentTotalIndex": 0,
    "positionX": 0,
    "positionY": 189,
    "height": 1605,
    "width": 1080,
    "checkable": false,
    "checked": false,
    "clickable": false,
    "enabled": true,
    "focusable": false,
    "focused": false,

```

```

    "longClickable": false,
    "scrollable": false,
    "selected": false,
    "password": false,
    "itemList": false,
    "calendarWindow": false,
    "relativeLocation": "Center",
    "idText": "android:id/content",
    "offset": 0,
    "drawTime": 0.0
  }, {
    "activity": "com.ichi2.anki.Info",
    "name": "android.widget.LinearLayout",
    "text": "",
    "contentDescription": "",
    "idXml": "",
    "componentIndex": 0,
    "componentTotalIndex": 0,
    "positionX": 0,
    "positionY": 189,
    "height": 1605,
    "width": 1080,
    "checkable": false,
    "checked": false,
    "clickable": false,
    "enabled": true,
    "focusable": false,
    "focused": false,
    "longClickable": false,
    "scrollable": false,
    "selected": false,
    "password": false,
    "itemList": false,
    "calendarWindow": false,
    "relativeLocation": "Center",
    "idText": "",
    "offset": 0,
    "drawTime": 0.0
  }, {
    "activity": "com.ichi2.anki.Info",
    "name": "android.widget.RelativeLayout",
    "text": "",
    "contentDescription": "",
    "idXml": "",
    "componentIndex": 0,
    "componentTotalIndex": 0,
    "positionX": 12,
    "positionY": 201,
    "height": 1450,
    "width": 1056,
    "checkable": false,
    "checked": false,
    "clickable": false,
    "enabled": true,
    "focusable": false,
    "focused": false,
    "longClickable": false,

```



```

    "scrollable": false,
    "selected": false,
    "password": false,
    "itemList": false,
    "calendarWindow": false,
    "relativeLocation": "Center",
    "idText": "",
    "offset": 0,
    "drawTime": 0.0
  }, {
    "activity": "com.ichi2.anki.Info",
    "name": "android.widget.FrameLayout",
    "text": "",
    "contentDescription": "",
    "idXml": "",
    "componentIndex": 0,
    "componentTotalIndex": 0,
    "positionX": 12,
    "positionY": 600,
    "height": 651,
    "width": 1056,
    "checkable": false,
    "checked": false,
    "clickable": false,
    "enabled": true,
    "focusable": false,
    "focused": false,
    "longClickable": false,
    "scrollable": false,
    "selected": false,
    "password": false,
    "itemList": false,
    "calendarWindow": false,
    "relativeLocation": "Center",
    "idText": "",
    "offset": 0,
    "drawTime": 0.0
  }, {
    "activity": "com.ichi2.anki.Info",
    "name": "android.webkit.WebView",
    "text": "",
    "contentDescription": "",
    "idXml": "",
    "componentIndex": 0,
    "componentTotalIndex": 0,
    "positionX": 12,
    "positionY": 600,
    "height": 651,
    "width": 1056,
    "checkable": false,
    "checked": false,
    "clickable": false,
    "enabled": true,
    "focusable": false,
    "focused": false,
    "longClickable": false,
    "scrollable": false,

```

```

    "selected": false,
    "password": false,
    "itemList": false,
    "calendarWindow": false,
    "relativeLocation": "Center",
    "idText": "",
    "offset": 0,
    "drawTime": 0.0
  }, {
    "activity": "com.ichi2.anki.Info",
    "name": "android.webkit.WebView",
    "text": "",
    "contentDescription": "",
    "idXml": "",
    "componentIndex": 0,
    "componentTotalIndex": 0,
    "positionX": 12,
    "positionY": 600,
    "height": 651,
    "width": 1057,
    "checkable": false,
    "checked": false,
    "clickable": false,
    "enabled": true,
    "focusable": true,
    "focused": true,
    "longClickable": false,
    "scrollable": true,
    "selected": false,
    "password": false,
    "itemList": false,
    "calendarWindow": false,
    "relativeLocation": "Center",
    "idText": "",
    "offset": 0,
    "drawTime": 0.0
  }, {
    "activity": "com.ichi2.anki.Info",
    "name": "android.view.View",
    "text": "",
    "contentDescription": "Welcome to AnkiDroid\n\nWith AnkiDroid, you can efficiently learn large amounts of vocabulary or any other material. AnkiDroid shows you the things to learn just before you forget them.\n\nTo get a first impression of how everything works, we recommend to start with a tutorial deck, which will give you some initial information, and try a bit out. Just touch \"Study\" after having opened it.\n\nCreate the tutorial deck now?",
    "idXml": "",
    "componentIndex": 0,
    "componentTotalIndex": 0,
    "positionX": 33,
    "positionY": 621,
    "height": 609,
    "width": 1015,

```

```

    "checkable": false,
    "checked": false,
    "clickable": false,
    "enabled": true,
    "focusable": false,
    "focused": false,
    "longClickable": false,
    "scrollable": false,
    "selected": false,
    "password": false,
    "itemList": false,
    "calendarWindow": false,
    "relativeLocation": "Center",
    "idText": "",
    "offset": 0,
    "drawTime": 0.0
  }, {
    "activity": "com.ichi2.anki.Info",
    "name": "android.widget.LinearLayout",
    "text": "",
    "contentDescription": "",
    "idXml": "com.ichi2.anki:id/info_buttons",
    "componentIndex": 1,
    "componentTotalIndex": 0,
    "positionX": 12,
    "positionY": 1662,
    "height": 120,
    "width": 1056,
    "checkable": false,
    "checked": false,
    "clickable": false,
    "enabled": true,
    "focusable": false,
    "focused": false,
    "longClickable": false,
    "scrollable": false,
    "selected": false,
    "password": false,
    "itemList": false,
    "calendarWindow": false,
    "relativeLocation": "Bottom",
    "idText": "com.ichi2.anki:id/info_buttons",
    "offset": 0,
    "drawTime": 0.0
  }, {
    "activity": "com.ichi2.anki.Info",
    "name": "android.widget.Button",
    "text": "Create tutorial",
    "contentDescription": "",
    "idXml": "com.ichi2.anki:id/info_tutorial",
    "componentIndex": 0,
    "componentTotalIndex": 0,
    "positionX": 12,
    "positionY": 1662,
    "height": 120,
    "width": 528,
    "checkable": false,

```

```

        "checked": false,
        "clickable": true,
        "enabled": true,
        "focusable": true,
        "focused": false,
        "longClickable": false,
        "scrollable": false,
        "selected": false,
        "password": false,
        "itemList": false,
        "calendarWindow": false,
        "relativeLocation": "Bottom left",
        "idText": "com.ichi2.anki:id/info_tutorial",
        "offset": 0,
        "drawTime": 0.0
    }, {
        "activity": "com.ichi2.anki.Info",
        "name": "android.widget.Button",
        "text": "Continue",
        "contentDescription": "",
        "idXml": "com.ichi2.anki:id/info_continue",
        "componentIndex": 1,
        "componentTotalIndex": 0,
        "positionX": 540,
        "positionY": 1662,
        "height": 120,
        "width": 528,
        "checkable": false,
        "checked": false,
        "clickable": true,
        "enabled": true,
        "focusable": true,
        "focused": false,
        "longClickable": false,
        "scrollable": false,
        "selected": false,
        "password": false,
        "itemList": false,
        "calendarWindow": false,
        "relativeLocation": "Bottom right",
        "idText": "com.ichi2.anki:id/info_continue",
        "offset": 0,
        "drawTime": 0.0
    }
    ]
}
}],
"app": {
    "name": "anki",
    "packageName": "com.ichi2.anki",
    "mainActivity": "com.ichi2.anki.DeckPicker",
    "version": "97",
}
}

```

---

Listing A.3: Example JSON format for MAGNETO execution

## Bibliography

- [1] N. Li and J. Offutt, “Test oracle strategies for model-based testing,” *IEEE Transactions on Software Engineering*, vol. 43, no. 4, pp. 372–395, April 2017.
- [2] B. W. Boehm, “Software engineering,” *IEEE Transactions on Computers*, vol. 25, no. 12, p. 1226–1241, December 1976.
- [3] S. Ng, T. Murnane, K. Reed, D. Grant, and T. Chen, “A preliminary survey on software testing practices in Australia,” in *Australian Software Engineering Conference*, vol. 2004, Victoria, Australia, February 2004, pp. 116–125.
- [4] V. Garousi and J. Zhi, “A survey of software testing practices in Canada,” *Sciencedirect The Journal of Systems and Software*, vol. 86, no. 5, p. 1354–1376, May 2013.
- [5] J. Dunn, “Career trends of software test automation engineering in 2019,” Online, February 2019, <https://dzone.com/articles/career-trends-of-software-test-automation-engineer>, last access August 2019.
- [6] S. Palamarchuk, “The true ROI of test automation,” Online, 2019, <https://abstracta.us/blog/test-automation/true-roi-test-automation/>, last access August 2019.
- [7] L. Koskela, *Test Driven: Practical TDD and Acceptance TDD for Java Developers*. Greenwich, CT: Manning Publications Company, 2008.
- [8] W. Platz, “Software fail watch: 5th edition,” Online white paper, 2017, <https://www.tricentis.com/resources/software-fail-watch-5th-edition/>, last access: August 2019.
- [9] MarketsAndMarkets, “Automation Testing Market by Component (Testing Types (Static Testing and Dynamic Testing) and Services), Endpoint Interface (Mobile, Web, Desktop, and Embedded Software), Organization Size, Vertical, and Region - Global Forecast to 2026,” <https://www.marketsandmarkets.com/Market-Reports/automation-testing-market-113583451.html>, 2022, [Online; accessed 08-March-2022].

- [10] D. Rafi, K. Moses, K. Petersen, and M. Mäntylä, “Benefits and limitations of automated software testing: Systematic literature review and practitioner survey,” in *International Workshop on Automation of Software Test*, June 2012, pp. 36–42.
- [11] K. Karhu, T. Repo, O. Taipale, and K. Smolander, “Empirical observations on software testing automation,” in *IEEE International Conference on Software Testing Verification, and Validation*, Colorado, USA, April 2009, pp. 201–209.
- [12] K. Wiklund, S. Eldh, D. Sundmark, and K. Lundqvist, “Impediments for software test automation: A systematic literature review: Impediments for software test automation,” *Wiley’s journal of Software Testing, Verification and Reliability*, vol. 27, no. 8, September 2017.
- [13] —, “Technical debt in test automation,” in *IEEE International Conference on Software Testing, Verification and Validation*, Montréal, Canada, April 2012, pp. 887–892.
- [14] A. Rendell, “Effective and pragmatic test driven development,” in *Agile 2008 Conference*, 2008, pp. 298–303.
- [15] G. Liebel, E. Alégroth, and R. Feldt, “State-of-practice in GUI-based system and acceptance testing: An industrial multiple-case study,” in *IEEE Euromicro Conference on Software Engineering and Advanced Applications*, Santander, Spain, September 2013, pp. 17–24.
- [16] S. Gruner and J. Zyl, “Software testing in small IT companies: A (not only) South African problem,” *South African Computer Journal*, vol. 47, pp. 7–32, November 2011.
- [17] L.-O. Damm, L. Lundberg, and D. Olsson, “Introducing test automation and test-driven development: An experience report,” *Sciencedirect Electronic Notes in Theoretical Computer Science*, vol. 116, pp. 3–15, January 2005.
- [18] P. Ammann and J. Offutt, *Introduction to Software Testing*, 2nd ed. Cambridge, UK: Cambridge University Press, 2017, ISBN 978-1107172012.
- [19] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, “iDFlakies: A framework for detecting and partially classifying flaky tests,” in *IEEE International Conference on Software Testing, Verification, and Validation*, Xi’an, China, April 2019, pp. 312–322.
- [20] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, “An empirical analysis of flaky tests,” in *ACM Foundations of Software Engineering*, Hong Kong, China, November 2014, pp. 643–653.
- [21] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, “DeFlaker: Automatically detecting flaky tests,” in *40th International Conference on Software Engineering*, Gothenburg, Sweden, May 2018.
- [22] K. Baral and J. Offutt, “An empirical analysis of blind tests,” in *IEEE Conference on Software Testing, Validation, and Verification*, Porto, Portugal, September 2020, p. 254–262.

- [23] K. Baral, J. Offutt, and F. Mulla, “Self determination: A comprehensive strategy for making automated tests more effective and efficient,” in *IEEE Conference on Software Testing, Validation, and Verification*, online, April 2021, pp. 127–136.
- [24] M. Pezze and M. Young, *Software Testing and Analysis: Process, Principles, and Techniques*. Hoboken, NJ: Wiley, 2008.
- [25] H. Krasner, “The Cost of Poor Software Quality in the US: A 2020 Report,” <https://www.it-cisq.org/the-cost-of-poor-software-quality-in-the-us-a-2020-report.htm>, 2020, [Online; accessed 18-March-2022].
- [26] R. S. Freedman, “Testability of software components,” *IEEE Transactions on Software Engineering*, vol. 17, no. 6, pp. 553–564, June 1991.
- [27] E. Beck and K. Gamma, “Test infected: Programmers love writing tests,” *Java Report*, vol. 3, no. 7, pp. 37–50, July 1998.
- [28] L. J. Morell, “A theory of error-based testing,” Ph.D. dissertation, University of Maryland, College Park, MD, USA, 1984, Technical report TR-1395.
- [29] ———, “A theory of error-based testing,” *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 844–857, August 1990.
- [30] J. Offutt, “Automatic test data generation,” Ph.D. dissertation, Georgia Institute of Technology, Atlanta, GA, USA, 1988, Technical report GIT-ICS 88/28.
- [31] R. A. DeMillo and J. Offutt, “Constraint-based automatic test data generation,” *IEEE Transaction on Software Engineering*, vol. 17, no. 9, pp. 900–910, September 1991.
- [32] D. J. Richardson, S. L. Aha, and T. O. O’Malley, “Specification-based test oracles for reactive systems,” in *ACM International Conference on Software Engineering*. Melbourne, Australia: ACM, May 1992, pp. 105–118.
- [33] W. E. Howden, “Theoretical and empirical studies of program testing,” *IEEE Transactions on Software Engineering*, vol. 4, no. 4, pp. 293–298, July 1978.
- [34] E. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, “The oracle problem in software testing: A survey,” *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, May 2015.
- [35] S. Analytics, “Strategy Analytics: Half the World Owns a Smartphone,” <https://www.businesswire.com/news/home/20210624005926/en/Strategy-Analytics-Half-the-World-Owns-a-Smartphone>, 2021, [Online; accessed 18-March-2022].
- [36] StatCounter, “Operating System Market Share Worldwide - February 2021,” <https://gs.statcounter.com/os-market-share>, 2021, [Online; accessed 08-March-2021].
- [37] “Google play store.” [Online]. Available: <https://play.google.com/store?hl=en>
- [38] “Apple app store.” [Online]. Available: <https://www.apple.com/ios/app-store/>

- [39] K. Karnes, “Why Users Uninstall Apps: 28% of People Feel Spammed [Survey],” <https://clevertap.com/blog/uninstall-apps/>, 2019, [Online; accessed 12-March-2021].
- [40] A. M. Memon, M. E. Pollack, and M. L. Soffa, “Using a goal-driven approach to generate test cases for GUIs,” in *IEEE International Conference on Software Engineering*, California, USA, May 1999, pp. 257–266.
- [41] “Nox emulator,” <https://www.bignox.com/>, [Online; accessed 14-June-2022].
- [42] “Genymotion emulator,” <https://www.genymotion.com>, [Online; accessed 14-June-2022].
- [43] “Android Emulator Documentation,” <http://developer.android.com/tools/help/emulator.html>, [Online; accessed 14-June-2022].
- [44] “Android uiautomator tool.” [Online]. Available: <https://developer.android.com/tools/testing-support-library/index.html#uia-viewer>
- [45] A. Bertolino, “Software testing research: Achievements, challenges, dreams,” in *IEEE Future of Software Engineering*, Minnesota, US, May 2007, pp. 85–103.
- [46] D. Ma’ayan, “The quality of JUnit tests: An empirical study report,” in *ACM/IEEE International Workshop on Software Qualities and their Dependencies*, Gothenburg, Sweden, May 2018, pp. 33–36.
- [47] M. Fowler, “Eradicating non-determinism in tests,” Online, 2011, <https://martinfowler.com/articles/nonDeterminism.html>, last accessed October 2019.
- [48] “Flakiness dashboard HOWTO,” Online, <http://www.chromium.org/developers/testing/flakiness-dashboard>, last access October 2019.
- [49] Anonymous, “TotT: Avoiding flakey tests,” Online, 2008, <https://testing.googleblog.com/2008/04/tott-avoiding-flakey-tests.html>, last accessed October 2019.
- [50] A. Vahabzadeh, A. M. Fard, and A. Mesbah, “An empirical study of bugs in test code,” in *IEEE International Conference on Software Maintenance and Evolution*, September 2015, pp. 101–110.
- [51] T. Xie, “Augmenting automatically generated unit-test suites with regression oracle checking,” in *European Conference on Object-Oriented Programming*, vol. 4067. Nantes, France: Springer, April 2006, pp. 380–403.
- [52] Y. Song, S. Thummalapenta, and T. Xie, “Unitplus: Assisting developer testing in eclipse,” in *OOPSLA Workshop on Eclipse Technology eXchange*. Montreal, Quebec, Canada: ACM, January 2007, pp. 26–30.
- [53] M. Staats, G. Gay, and M. P. E. Heimdahl, “Automated oracle creation support, or: how I learned to stop worrying about fault propagation and love mutation testing,” in *IEEE International Conference on Software Engineering*, Piscataway, NJ, USA, 2012, pp. 870–880.



- [54] P. Loyola, M. Staats, I.-Y. Ko, and G. Rothermel, “Dodona: Automated oracle data set selection,” in *International Symposium on Software Testing and Analysis*. San Jose, CA, USA: ACM, July 2014, pp. 193–203.
- [55] D. Schuler and A. Zeller, “Checked coverage: An indicator for oracle quality,” *Wiley’s Software Testing, Verification, and Reliability*, vol. 23, no. 7, pp. 531–551, 2013.
- [56] J. Zhi and V. Garousi, “On adequacy of assertions in automated test suites: An empirical investigation,” in *IEEE International Workshop on Regression Testing*, March 2013, pp. 382–391.
- [57] Q. Xie and A. Memon, “Designing and comparing automated test oracles for GUI-based software applications,” *ACM Transaction on Software Engineering and Methodology*, vol. 16, no. 1, February 2007.
- [58] M. Staats, M. W. Whalen, and M. P. Heimdahl, “Programs, tests, and oracles: The foundations of testing revisited,” in *International Conference on Software Engineering*. Hawaii, US: ACM, May 2011, pp. 391–400.
- [59] K. Koster and D. C. Kao, “State coverage: A structural test adequacy criterion for behavior checking,” in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, September 2007, pp. 541–544.
- [60] S. Yoo and M. Harman, “Regression testing minimization, selection and prioritization: A survey,” *Wiley’s Software Testing, Verification, and Reliability*, vol. 22, no. 2, pp. 67–120, March 2012.
- [61] T. Y. Chen and M. F. Lau, “Dividing strategies for the optimization of a test suite,” *Information Processing Letters, Elsevier*, vol. 60, pp. 135–141, November 1996.
- [62] P. Ammann, M. E. Delamaro, and J. Offutt, “Establishing theoretical minimal sets of mutants,” in *IEEE International Conference on Software Testing, Verification, and Validation*, Ohio, US, March 2014, pp. 21–30.
- [63] B. Vaysburg, L. H. Tahat, and B. Korel, “Dependence analysis in reduction of requirement based test suites,” *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 4, p. 107–111, July 2002.
- [64] R. Gupta, M. J. Harrold, and M. L. Soffa, “An approach to regression testing using slicing,” in *IEEE International Conference on Software Maintenance*, vol. 92, Florida, US, November 1992, pp. 299–308.
- [65] M. J. Harrold and M. L. Soffa, “Interprocedural data flow testing,” *ACM SIGSOFT Software Engineering Notes*, vol. 14, no. 8, pp. 158–167, November 1989.
- [66] —, “An incremental approach to unit testing during maintenance,” in *IEEE International Conference on Software Maintenance*, California, US, October 1988, pp. 362–367.

- [67] S.-S. Yau and Z. Kishimoto, "Method for revalidating modified programs in the maintenance phase." in *IEEE International Computer Software & Applications Conference*, July 1987, pp. 272–277.
- [68] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. A. London, "Incremental regression testing," in *IEEE Conference on Software Maintenance*, Quebec, Canada, September 1993, pp. 348–357.
- [69] G. Rothermel and M. J. Harrold, "A safe, efficient algorithm for regression test selection," in *IEEE Conference on Software Maintenance*, Quebec, Canada, September 1993, pp. 358–367.
- [70] —, "Selecting tests and identifying test coverage requirements for modified software," in *ACM SIGSOFT International Symposium on Software Testing and Analysis*, Seattle, Washington, USA, August 1994, pp. 169–184.
- [71] —, *Efficient, effective regression testing using safe test selection techniques*. US: Clemson University, 1996.
- [72] —, "A safe, efficient regression test selection technique," *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 2, pp. 173–210, April 1997.
- [73] F. I. Vokolos and P. G. Frankl, "Pythia: A regression test selection tool based on textual differencing," in *Reliability, Quality and Safety of Software-Intensive Systems*. Springer, 1997, pp. 3–21.
- [74] F. Vokolos and P. Frankl, "Empirical evaluation of the textual differencing regression testing technique," in *IEEE International Conference on Software Maintenance*, Maryland, US, November 1998, pp. 44–53.
- [75] Y.-F. Chen, D. Rosenblum, and K.-P. Vo, "Testtube: A system for selective regression testing," in *IEEE International Conference on Software Engineering*, Sorrento, Italy, May 1994, pp. 211–220.
- [76] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 10, pp. 929–948, 2001.
- [77] Y. Lou, D. Hao, and L. Zhang, "Mutation-based test-case prioritization in software evolution," in *IEEE International Symposium on Software Reliability Engineering*, November 2015, pp. 46–57.
- [78] L. Zhang, D. Marinov, and S. Khurshid, "Faster mutation testing inspired by test prioritization and reduction," in *ACM International Symposium on Software Testing and Analysis*, Lugano, Switzerland, July 2013, pp. 235–245.
- [79] H. Do and G. Rothermel, "On the use of mutation faults in empirical assessments of test case prioritization techniques," *IEEE Transactions on Software Engineering*, vol. 32, no. 9, pp. 733–752, 2006.

- [80] R. N. Zaeem, M. R. Prasad, and S. Khurshid, “Automated generation of oracles for testing user-interaction features of mobile apps,” in *IEEE International Conference on Software Testing, Verification, and Validation*, Ohio, USA, 2014, pp. 183–192.
- [81] Z. Liu, “Discovering UI display issues with visual understanding,” in *IEEE/ACM International Conference on Automated Software Engineering*, Melbourne, Australia, September 2020, pp. 1373–1375.
- [82] Š. Packevičius, D. Barisas, A. Ušaniov, E. Guogis, and E. Bareiša, “Text semantics and layout defects detection in Android apps using dynamic execution and screenshot analysis,” in *Information and Software Technologies*. Cham, Denmark: Springer, August 2018, pp. 279–292.
- [83] T. Su, Y. Yan, J. Wang, J. Sun, Y. Xiong, G. Pu, K. Wang, and Z. Su, “Fully automated functional fuzzing of Android apps for detecting non-crashing logic bugs,” in *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2021, pp. 1–31.
- [84] O. Riganelli, S. P. Mottadelli, C. Rota, D. Micucci, and L. Mariani, “Data loss detector: automatically revealing data loss bugs in Android apps,” in *ACM SIGSOFT International Symposium on Software Testing and Analysis*, July 2020, pp. 141–152.
- [85] C.-Z. Yang, C.-J. Lai, P. Lu, and Z.-J. You, “LAD: A layout anomaly detector for Android applications,” in *International Conference on Software Engineering and Knowledge Engineering*, July 2019, pp. 557–728.
- [86] M. M. Eler, J. M. Rojas, Y. Ge, and G. Fraser, “Automated accessibility testing of mobile apps,” in *IEEE International Conference on Software Testing, Verification, and Validation*, Vasteras, Sweden, April 2018, pp. 116–126.
- [87] Y. Kou, H. Jeong, and E. Lee, “Image-based bug oracle automation for bug report reproduction using wt detection,” in *IEEE International Conference on Software Engineering and Artificial Intelligence*, Xiamen, China, June 2021, pp. 43–47.
- [88] Y. Koroglu and A. Sen, “Reinforcement learning-driven test generation for Android GUI applications using formal specifications,” *arXiv preprint arXiv:1911.05403*, 2019.
- [89] E. T.-H. Chu and J.-Y. Lin, “Automated GUI testing for Android news applications,” in *IEEE International Symposium on Computer, Consumer and Control*, Taichung, Taiwan, December 2018, pp. 14–17.
- [90] J. Brown, Z. Zhou, and Y.-W. Chow, “Metamorphic testing of navigation software: A pilot study with Google Maps,” in *Hawaii International Conference on System Sciences*, Hawaii, US, January 2018, pp. 1–10.
- [91] A. Méndez-Porrás, G. Méndez-Marín, A. Tablada-Rojas, M. N. Hidalgo, J. M. García-Chamizo, M. Jenkins, and A. Martínez, “A distributed bug analyzer based on user-interaction features for mobile apps,” *Journal of Ambient Intelligence and Humanized Computing*, vol. 8, no. 4, pp. 579–591, 2017.

- [92] Š. Packevičius, A. Ušaniov, Š. Stanskis, and E. Bareiša, “The testing method based on image analysis for automated detection of UI defects intended for mobile applications,” in *International Conference on Information and Software Technologies*, vol. 458. Springer, January 2015, pp. 560–576.
- [93] J. Sun, T. Su, J. Li, Z. Dong, G. Pu, T. Xie, and Z. Su, “Understanding and finding system setting-related defects in Android apps,” in *ACM SIGSOFT International Symposium on Software Testing and Analysis*, Aarhus, Denmark, July 2021, pp. 204–215.
- [94] R. Jabbarvand, F. Mehralian, and S. Malek, “Automated construction of energy test oracles for Android,” in *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Athens, Greece, November 2020, pp. 927–938.
- [95] M. C. Júnior, D. Amalfitano, L. Garcés, A. R. Fasolino, S. A. Andrade, and M. Delamaro, “Dynamic testing techniques of non-functional requirements in mobile apps: A systematic mapping study,” *ACM Computing Surveys*, vol. 54, pp. 1–38, September 2022.
- [96] L. Mariani, M. Pezzè, and D. Zuddas, “Augusto: Exploiting popular functionalities for the generation of semantic GUI tests with oracles,” in *International Conference on Software Engineering*, Gothenburg, Sweden, May 2018, pp. 280–290.
- [97] A. M. Memon, M. E. Pollack, and M. L. Soffa, “Automated test oracles for GUIs,” *ACM SIGSOFT Software Engineering Notes*, vol. 25, no. 6, pp. 30–39, November 2000.
- [98] M. I. Malik, M. A. Sindhu, A. S. Khattak, R. A. Abbasi, and K. Saleem, “Automating test oracles from restricted natural language agile requirements,” *Wiley’s Expert Systems*, vol. 38, no. 1, January 2021.
- [99] A. Goffi, A. Gorla, M. D. Ernst, and M. Pezzè, “Automatic generation of oracles for exceptional behaviors,” in *ACM International Symposium on Software Testing and Analysis*, Saarbrücken, Germany, July 2016, pp. 213–224.
- [100] W. Platz, “As test automation matures, so do false positives,” Online, <https://www.stickyminds.com/article/test-automation-matures-so-do-false-positives>, last access November 2022.
- [101] M. Staats, M. W. Whalen, and M. P. E. Heimdahl, “Better testing through oracle selection,” in *ACM International Conference on Software Engineering (NIER Track)*, Hawaii, US, May 2011, pp. 892–895.
- [102] K. Baral and J. Offutt, “Blind test online appendix <https://github.com/Keshina/BlindTest>.”
- [103] Google, “Google test automation conference,” Online, <https://developers.google.com/google-test-automation-conference/>, last access: September 2019.

- [104] P. Copeland, “Google’s innovation factory (keynote address),” in *IEEE Conference on Software Testing, Validation, and Verification*, March 2010.
- [105] C. L. Goues, M. Pradel, and A. Roychoudhury, “Automated program repair,” *Communications of the ACM*, vol. 62, no. 12, p. 56–65, November 2019.
- [106] L. Gazzola, D. Micucci, and L. Mariani, “Automatic software repair: A survey,” *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 34–67, January 2019.
- [107] S. Eldh, J. Brandt, M. Street, H. Hansson, and S. Punnekkat, “Towards fully automated test management for large complex systems,” in *IEEE International Conference on Software Testing, Verification, and Validation*, Paris, France, April 2010, pp. 412–420.
- [108] E. Enoiu and M. Frasher, “Test agents: The next generation of test cases,” in *IEEE International Conference on Software Testing, Verification, and Validation Workshop - NEXt level of Test Automation (NEXTA)*, Xian, China, April 2019, pp. 305–308.
- [109] J. Offutt, “From spec-based testing to test automation and beyond (keynote address),” in *Workshop on Advances in Model Based Testing (A-MOST)*, Vasteros, Sweden, April 2018. [Online]. Available: <https://cs.gmu.edu/~offutt/documents/slides/2018AMost-keynote.pptx>
- [110] —, “It is great that we automate our tests, but why are they so bad?” SAST Industry Day at the 11th IEEE Conference on Software Testing, Validation, and Verification, Vasteros, Sweden, April 2018. [Online]. Available: <https://cs.gmu.edu/~offutt/documents/slides/2018SAST-ICST.pptx>
- [111] K. Baral, F. Mulla, and J. Offutt, “Smart test online appendix <https://github.com/Keshina/smartTest>.”
- [112] S. Elbaum, A. G. Malishevsky, and G. Rothermel, “Prioritizing test cases for regression testing,” *SIGSOFT Software Engineering Notes*, vol. 25, no. 5, pp. 102–112, 2000.
- [113] G. Rothermel, S. Elbaum, A. Malishevsky, P. Kallakuri, and B. Davia, “The impact of test suite granularity on the cost-effectiveness of regression testing,” in *ACM International Conference on Software Engineering*, Florida, US, May 2002, pp. 130–140.
- [114] H. Do, G. Rothermel, and A. Kinneer, “Empirical studies of test case prioritization in a JUnit testing environment,” in *IEEE International symposium on software reliability engineering*, Naples, Italy, November 2004, pp. 113–124.
- [115] H. Do and G. Rothermel, “A controlled experiment assessing test case prioritization techniques via mutation faults,” in *IEEE International Conference on Software Maintenance*, Budapest, Hungary, September 2005, pp. 411–420.
- [116] Anonymous, “Asm,” Online, <https://asm.ow2.io/>, last access: May 2020.
- [117] S. M. Ghaffarian and H. R. Shahriari, “Neural software vulnerability analysis using rich intermediate graph representations of programs,” *Elsevier’s Information Sciences*, pp. 189–207, April 2021.

- [118] Graphviz, “The DOT language,” Online, <https://www.graphviz.org/doc/info/lang.html>, last access: May 2020.
- [119] D. Crockford, “Introducing JSON,” Online, <https://www.json.org/json-en.html> , last access: May 2020.
- [120] P. S. Foundation, “Sequencematcher,” Online, <https://docs.python.org/3/library/difflib.html#difflib>, last access: May 2020.
- [121] ———, “difflib — helpers for computing deltas,” Online, <https://docs.python.org/3/library/difflib.html>, last access: May 2020.
- [122] T. A. S. Foundation, “Apache maven project,” Online, <https://maven.apache.org/what-is-maven.html>, last access: May 2020.
- [123] R. Just, F. Schweiggert, and G. M. Kapfhammer, “MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler,” in *International Conference on Automated Software Engineering*, Kansas, US, November 2011, pp. 612–615.
- [124] J. H. Andrews, L. C. Briand, and Y. Labiche, “Is mutation an appropriate tool for testing experiments?” in *IEEE International Conference on Software Engineering*, Missouri, US, May 2005, pp. 402–411.
- [125] H. Coles, “PIT mutation testing tool,” Online, <https://github.com/hcoles/pitest>, last access: October 2020.
- [126] M. J. Harrold and M. L. Soffa, “Selecting and using data for integration testing,” *IEEE Software*, vol. 8, no. 2, pp. 58–65, March 1991.
- [127] V. Martena, A. Orso, and M. Pezzé, “Interclass testing of object oriented software,” in *IEEE International Conference on Engineering of Complex Computer Systems*, Maryland, US, December 2002, pp. 135–144.
- [128] Statista, “Number of apps available in leading app stores,” 2022. [Online]. Available: <https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/>
- [129] ———, “Installed base of smartphones by operating system from 2015 to 2017,” 2022. [Online]. Available: <https://www.statista.com/statistics/385001/smartphone-worldwide-installed-base-operating-systems/>
- [130] A. K. Maji, K. Hao, S. Sultana, and S. Bagchi, “Characterizing failures in mobile OSes: A case study with Android and Symbian,” in *IEEE International Symposium on Software Reliability Engineering*, California, US, November 2010, pp. 249–258.
- [131] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, and Z. Su, “Practical GUI testing of Android applications via model abstraction and refinement,” in *IEEE International Conference on Software Engineering*, May 2019, p. 269–280.
- [132] K. Mao, M. Harman, and Y. Jia, “Sapienz: Multi-objective automated testing for Android applications,” in *ACM International Symposium on Software Testing and Analysis*, Saarbrücken, Germany, July 2016, pp. 94–105.

- [133] N. P. Borges Jr., J. Hotzkow, and A. Zeller, *DroidMate-2: A Platform for Android Test Generation*. Montpellier, France: Association for Computing Machinery, September 2018, p. 916–919.
- [134] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk, “Automatically discovering, reporting and reproducing Android application crashes,” in *IEEE International Conference on Software Testing, Verification, and Validation*, April 2016, pp. 33–44.
- [135] L. Deng, J. Offutt, P. Ammann, and N. Mirzaei, “Mutation operators for testing Android apps,” *Elsevier’s Information and Software Technology, special issue from the mutation 2015 workshop*, vol. 81, pp. 154–168, January 2017.
- [136] X. Zeng, D. Li, W. Zheng, F. Xia, Y. Deng, W. Lam, W. Yang, and T. Xie, “Automated test input generation for Android: Are we really there yet in an industrial case?” in *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, November 2016, p. 987–992.
- [137] P. Kong, L. Li, J. Gao, K. Liu, T. F. Bissyandé, and J. Klein, “Automated testing of Android apps: A systematic literature review,” *IEEE Transactions on Reliability*, vol. 68, no. 1, pp. 45–66, 2019.
- [138] J. Johanson, J. Mahmud, T. Wendland, K. Moran, J. Rubin, and M. Fazzini, “An empirical investigation into the reproduction of bug reports for Android apps,” in *IEEE International Conference on Software Analysis, Evolution and Reengineering*, Honolulu, Hawaii, March 2022.
- [139] Y. Zhao, T. Yu, T. Su, Y. Liu, W. Zheng, J. Zhang, and W. G. J. Halfond, “Re-droid: Automatically reproducing Android application crashes from bug reports,” in *IEEE/ACM International Conference on Software Engineering*, Montreal, Canada, May 2019, pp. 128–139.
- [140] M. Fazzini, M. Prammer, M. d’Amorim, and A. Orso, “Automatically translating bug reports into test cases for mobile apps,” in *ACM SIGSOFT International Symposium on Software Testing and Analysis*, Amsterdam, Netherlands, July 2018, pp. 141–152.
- [141] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, “API change and fault proneness: A threat to the success of Android apps,” in *ACM Joint Meeting on Foundations of Software Engineering*, Saint Petersburg, Russia, August 2013, pp. 477–487.
- [142] G. Bavota, M. Linares-Vásquez, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, “The impact of API change- and fault-proneness on the user ratings of Android apps,” *IEEE Transactions on Software Engineering*, vol. 41, no. 4, pp. 384–407, April 2015.
- [143] D. Han, C. Zhang, X. Fan, A. Hindle, K. Wong, and E. Stroulia, “Understanding Android fragmentation with topic analysis of vendor-specific bugs,” in *IEEE Working Conference on Reverse Engineering*, Washington DC, USA, 2012, pp. 83–92.

- [144] L. Wei, Y. Liu, and S. C. Cheung, “Taming Android fragmentation: Characterizing and detecting compatibility issues for Android apps,” in *IEEE/ACM International Conference on Automated Software Engineering*, Singapore, September 2016, pp. 226–237.
- [145] “Android fragmentation statistics,” 2014. [Online]. Available: <http://opensignal.com/reports/2014/android-fragmentation/>
- [146] M. Fazzini and A. Orso, “Automated cross-platform inconsistency detection for mobile apps,” in *IEEE/ACM International Conference on Automated Software Engineering*, Urbana, US, October 2017, pp. 308–318.
- [147] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, “Atrina: Inferring unit oracles from GUI test cases,” in *IEEE International Conference on Software Testing, Verification, and Validation*, Chicago, US, April 2016, pp. 330–340.
- [148] K. Chen, Y. Li, Y. Chen, C. Fan, Z. Hu, and W. Yang, “GLIB: Towards automated test oracle for graphically-rich applications,” in *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Athens, Greece, August 2021, pp. 1093–1104.
- [149] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo, “Understanding the test automation culture of app developers,” in *IEEE International Conference on Software Testing, Verification, and Validation*, Graz, Austria, April 2015, pp. 1–10.
- [150] M. B. Miles, A. M. Huberman, and J. Saldaña, *Qualitative data analysis: a methods sourcebook*. Thousand Oaks, California: SAGE Publications, Inc., 2013.
- [151] K. Charmaz, *Constructing Grounded Theory*. SAGE Publications Inc., 2006.
- [152] K. Baral, J. Johnson, J. Mahmud, S. Salma, M. Fazzini, J. Rubin, J. Offutt, and K. Moran, “Magneto online appendix <https://github.com/SageSELab/Magneto>.”
- [153] M. Linares-Vásquez, G. Bavota, M. Tufano, K. Moran, M. Di Penta, C. Vendome, C. Bernal-Cárdenas, and D. Poshyvanyk, “Enabling mutation testing for Android apps,” in *ACM Foundations of Software Engineering*, September 2017, p. 233–244.
- [154] K. Moran, B. Li, C. Bernal-Cárdenas, D. Jelf, and D. Poshyvanyk, “Automated reporting of GUI design violations in mobile apps,” in *IEEE International Conference on Software Engineering Companion*, Gothenburg, Sweden, May 2018.
- [155] K. P. Moran, C. Bernal-Cardenas, M. Curcio, R. Bonett, and D. Poshyvanyk, “Machine learning-based prototyping of graphical user interfaces for mobile apps,” *IEEE Transactions on Software Engineering*, vol. 46, pp. 196–221, 2018.
- [156] W. O. Galitz, *The Essential Guide to User Interface Design: An Introduction to GUI Design Principles and Techniques*. USA: John Wiley & Sons, Inc., 2007.
- [157] Z. Wang, A. Bovik, H. Sheikh, and E. Simoncelli, “Image quality assessment: from error visibility to structural similarity,” *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, 2004.



- [158] CIE, “International Commission on Illumination,” Online, <http://cie.co.at/>, last access March 2022.
- [159] G. Sharma and R. Bala, *Digital color imaging handbook*. CRC press, 2017.
- [160] W. Backhaus, R. Kliegl, and J. S. Werner, *Color vision: Perspectives from different disciplines*. Walter de Gruyter, 1998.
- [161] S. Hoffstaetter, “Python tesseract,” Online, <https://pypi.org/project/pytesseract/>, last access March 2022.
- [162] Polyglot, “Polyglot.” [Online]. Available: <https://polyglot.readthedocs.io/en/latest/index.html>, last access March 2022
- [163] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” *Neural Information Processing Systems*, vol. 60, p. 84–90, June 2012.
- [164] D. D. D. Group, “Rico dataset - interaction mining,” <https://interactionmining.org/rico>, [Online; accessed 08-March-2021].
- [165] E. W. Forgy, “Cluster analysis of multivariate data : efficiency versus interpretability of classifications,” *Biometrics*, vol. 21, pp. 768–769, 1965.
- [166] Heartex, “Label studio,” Online, <https://labelstud.io/>, last access March 2022.
- [167] O. Chaparro, C. Bernal-Cárdenas, J. Lu, K. Moran, A. Marus, M. Di Penta, D. Poshyanyk, and V. Ng, “Assessing the quality of the steps to reproduce in bug reports,” in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Bergamo, Italy, August 2019, pp. 86–96.
- [168] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, “Guided, stochastic model-based GUI testing of Android apps,” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 245–256. [Online]. Available: <http://doi.acm.org/10.1145/3106237.3106298>
- [169] Y. Su, Z. Liu, C. Chen, J. Wang, and Q. Wang, “Owleyes-online: A fully automated platform for detecting and localizing UI display issues,” in *ACM European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Athens, Greece, August 2021, p. 1500–1504.
- [170] B. Yang, Z. Xing, X. Xia, C. Chen, D. Ye, and S. Li, “Don’t do that! Hunting down visual design smells in complex UIs against design guidelines,” in *IEEE International Conference on Software Engineering*, Madrid, Spain, May 2021, p. 761–772.
- [171] D. Zhao, Z. Xing, C. Chen, X. Xu, L. Zhu, G. Li, and J. Wang, “Seenomaly: Vision-based linting of GUI animation effects against design-don’t guidelines,” in *ACM/IEEE International Conference on Software Engineering*, Seoul, South Korea, June 2020, p. 1286–1297.

- [172] K. Baral, R. Mohod, J. Flamm, S. Goldrich, and P. Ammann, “Evaluating a test automation decision support tool,” in *IEEE International Conference on Software Testing, Verification, and Validation Workshop - TAICPART*, Xi’an, China, April 2019, pp. 69–76.
- [173] J. Offutt, B. Lindström, and K. Baral, “Teaching an international distributed discussion-based course,” in *International Conference on Frontiers in Education: Computer Science and Computer Engineering*, Las Vegas, US, August 2019, pp. 149–154.
- [174] K. Baral and P. Ammann, “Teaching a testing concept (junit) with active learning,” in *IEEE International Conference on Software Testing, Verification and Validation Workshops - TestEd*, Porto, Portugal, October 2020, pp. 411–411.
- [175] M. L. Chaim, K. Baral, J. Offutt, M. Concilio, and R. P. A. Araujo, “Efficiently finding data flow subsumptions,” in *IEEE Conference on Software Testing, Verification, and Validation*, online, April 2021, pp. 94–104.
- [176] K. Baral, J. Offutt, P. Ammann, and R. Mohod, “Practice makes better: Quiz retake software to increase student learning,” in *ACM International Workshop on Education through Advanced Software Engineering and Artificial Intelligence*, Athens, Greece, 2021, p. 47–53.
- [177] Y. Zhao, S. Talebipour, K. Baral, H. Park, L. Yee, S. A. Khan, Y. Brun, N. Medvidović, and K. Moran, “Avgust: automating usage-based test generation from videos of app executions,” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, November 2022, pp. 421–433.
- [178] J. Offutt and K. Baral, “Designing divergent thinking, creative problem solving exams,” in *IEEE International Conference on Software Engineering: Software Engineering Education and Training*, May 2022, pp. 82–89.

## Curriculum Vitae

Kesina Baral received her Bachelors in Computer Engineering from Tribhuvan University, Nepal in 2015. She received her Masters in Software Engineering from George Mason University in 2021. Her doctoral work focuses on the challenges of test automation, specifically, test oracle problem and test suite maintenance. Over the course of her PhD, she has published papers in IEEE Conference on Software Testing, Verification and Validation (ICST), Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), and International Conference on Software Engineering (ICSE), among others.