UNIVERSITY OF CALIFORNIA,
IRVINE

Advancing Automated Software Testing Through Test Reuse

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Software Engineering

by

Jun-Wei Lin

Dissertation Committee:
Professor Sam Malek, Chair
Associate Professor James A. Jones
Professor Cristina Videira Lopes

2021

DEDICATION

To my beloved wife and daughter, Wan-Ting and Ariel.

Contents

Page

LIST OF FIGURES **vi**

LIST OF TABLES **vii**

LIST OF ALGORITHMS **ix**

ACKNOWLEDGMENTS **x**

VITA **xi**

ABSTRACT OF THE DISSERTATION **xiii**

1 Introduction **1**
- 1.1 Dissertation Overview 4
- 1.2 Dissertation Structure 5

2 Related Work and Research Gap **7**
- 2.1 Related Work 7
  - 2.1.1 Common Functionalities Across GUI-Based Apps 8
  - 2.1.2 Intra-Platform Test Transfer 8
  - 2.1.3 Cross-Platform Testing 9
  - 2.1.4 Test Augmentation 10
  - 2.1.5 Empirical studies on Mobile App Testing 12
- 2.2 Research Gap 13

3 Research Problem **15**
- 3.1 Problem Statement 15
- 3.2 Thesis Statement 16
- 3.3 Research Hypotheses 16

4 Intra-Platform Test Transfer **19**
- 4.1 Introduction 20
- 4.2 Motivating Example 22
- 4.3 Approach Overview 24
- 4.4 Test Augmentation 26

4.5 Model Extraction 26
4.6 Test Generation 29
4.6.1 Computing Similarity Score 31
4.6.2 Reachability Check 34
4.6.3 Actions for the Transferred GUI and Oracle Events 35
4.6.4 Termination Criteria 37
4.7 Evaluation 38
4.7.1 Experimental Setup 38
4.7.2 RQ1: Effectiveness 41
4.7.3 RQ2: Factors Impacting Effectiveness 43
4.7.4 RQ3: Efficiency 44
4.7.5 RQ4: Factors Impacting Efficiency 45
4.7.6 Threats to Validity 45
4.8 Conclusion and Future Work 46

5 Inter-Platform Test Transfer 47
5.1 Introduction 48
5.2 Background and Motivating Example 51
5.3 Approach 54
5.3.1 Context Extraction 54
5.3.2 Action Transformation 56
5.3.3 NavGraph Extraction 57
5.4 Test Generation 59
5.4.1 Similarity Computation 60
5.4.2 Reachability Check 62
5.4.3 Action Generation 63
5.4.4 Walk-Through of the Motivating Example 64
5.5 Evaluation 65
5.5.1 Experimental Setup 66
5.5.2 RQ1: Effectiveness 68
5.5.3 RQ2: Factors Impacting Effectiveness 69
5.5.4 RQ3: Reduced Effort 71
5.5.5 RQ4: Efficiency 72
5.6 Assumptions and Limitations 73
5.7 Conclusion 74

6 Feature-Based Test Augmentation 75
6.1 Introduction 76
6.2 Background 80
6.3 Approach 81
6.3.1 Base Path Construction 82
6.3.2 App Exploration 84
6.4 Test Generation 86
6.4.1 Computing path difference 89
6.5 Evaluation 90

6.5.1 Experimental Setup 90
6.5.2 RQ1: Quality of the Generated Tests 92
6.5.3 RQ2: Fault Detection Effectiveness 96
6.5.4 RQ3: Influence of the Independent Variables 97
6.6 Conclusion 98

7 Test Automation in Open-Source Android Apps: A Large-Scale Empirical Study 100
7.1 Introduction 101
7.2 Test Automation in Android 104
7.2.1 Unit and UI Tests 104
7.2.2 The Testing Pyramid Practice 105
7.3 Methodology 106
7.3.1 Study Subjects and Selection Criteria 107
7.3.2 Data Collection and Analysis 109
7.3.3 Survey 111
7.4 Results 113
7.4.1 Prevalence and Trends in the Adoption of Test Automation 113
7.4.2 Working Habits of Mobile App Developers 120
7.4.3 Association Between Presence of Automated Tests and Popularity . . 124
7.5 Threats to Validity 129
7.6 Conclusion 130

8 Conclusion 131
8.1 Research Contributions 132
8.2 Future Work 133

Bibliography 135

List of Figures

Page

4.1 Overview of CraftDroid 24
4.2 Excerpted registration process on Rainbow Shops 25
4.3 Excerpted registration process on Yelp 25
4.4 Excerpted UI Transition Graph for Yelp 27

5.1 Saving a draft with the web app of WordPress 52
5.2 Saving a draft with the Android app of WordPress 52
5.3 Overview of TransDroid 54
5.4 Retrieved contexts of (“Posts”, *mouseOver*) in Table 5.1 56
5.5 Excerpted Navigation Graph for WordPress on Android 58
5.6 Different interaction flows to create a page on Dokuwiki across platforms . . 70

6.1 Different use-case scenarios for timetable management in the School Planner app. 78
6.2 Example of states and paths in the School Planner app shown in Figure 6.1 80
6.3 Overview of Route 82
6.4 Examples of valid and invalid paths considered by `getCyclicPaths()` in Algorithm 7 88
6.5 Deleting a checked item in Shopping List with different controllers 93
6.6 Creating a note in Writeily Pro with different input data 94
6.7 View photo details in OwnCloud with different input data 95
6.8 Timetable management in School Planner with different control flows 96

7.1 Example Test Classes Generated by Android Studio 105
7.2 Illustration of the Testing Pyramid Practice from [68] 106
7.3 Flow of Data Collection and Analysis in This Study 107
7.4 Distribution of the Google Play Apps by Category 111
7.5 Prevalence of Test Automation of the Google Play Apps by Category 117
7.6 Days to Add the First Test after Project Creation 118
7.7 Percentage of Repositories Containing Tests over the Past 5 Years 119

List of Tables

Page

4.1 Types of oracle supported by CraftDroid. (w'_i, a_i): the source oracle event. (w_n, a_n): the transferred target oracle event. 36
4.2 Subject apps. 39
4.3 Test cases for the proposed functionalities. 40
4.4 Identified main functionalities for subject apps. 41
4.5 Effectiveness and Efficiency Evaluation of CraftDroid. 42
4.6 Pearson correlation coefficient between average test length and effectiveness. 43

5.1 The GUI and oracle events for saving a draft in WordPress on different platforms 53
5.2 Action Transformation Rules in TransDroid 57
5.3 Assertion types supported by TransDroid. (w_i, a_i): source oracle event. (w_n, a_n): transferred target event. 64
5.4 Subject apps and test suites 66
5.5 Effectiveness evaluation of TransDroid 68
5.6 Average reduction of effort 71
5.7 Efficiency evaluation of TransDroid 72

6.1 Subject tests and experimental results 91
6.2 Categories of tests generated by Route 96
6.3 Mutants created for each subject apps 97
6.4 Percentage of mutants killed by the original tests and augmented tests with different lookahead step k and number of generated tests n 97

7.1 Distribution of Apps by App Market* 110
7.2 Distribution of Apps by Year Created 110
7.3 Distribution of Apps in Terms of Presence of Test Cases 113
7.4 The Ways of Testing Android Apps by the Survey Participants 115
7.5 The Reasons for not Adopting Test Automation by the Survey Participants . 115
7.6 The Biggest Challengs in Testing Android Apps by the Survey Participants . 116
7.7 The Most Important or Useful Criteria for Evaluating Android App Tests by the Survey Participants 116
7.8 Probability of Observing Consistent Behavior on the Apps by the Same Developers. S_a: Clusters of Apps by the Same Developers. S_b: by Different Developers 121

7.9 The Reasons for the Preference of Unit Testing by the Survey Participants . 122
7.10 Distribution of the Number of Tests in the Apps with Both Types of Tests. 1Q: 1st quartile. 2Q: 2nd quartile (median). 3Q: 3rd quartile. 123
7.11 Distribution of the Popularity and Satisfaction Metrics of the Apps. 1Q: 1st quartile. 2Q: 2nd quartile (median). 3Q: 3rd quartile. 124
7.12 Impact of Having Tests on the Popularity of Apps. R_w: Apps with Tests. R_o: Apps Without Tests. 126
7.13 Impact of Having Tests on the User Satisfaction of Apps. R_w: Apps with Tests. R_o: Apps Without Tests. 128

LIST OF ALGORITHMS

Page

1 CraftDroid: Test Augmentation 27
2 CraftDroid: Test Generation 30
3 TransDroid: Test Generation 61
4 TransDroid: Function: isReachable() 63
5 Route: Base Path Construction 83
6 Route: App Exploration 85
7 Route: Test Generation 87
8 Route: Path Difference 89

ACKNOWLEDGMENTS

First and foremost, my greatest gratitude goes to my advisor Dr. Sam Malek who mentors my professional and personal growth. For all the good things that come with this degree I owe them to Sam. Words also cannot describe how grateful I am to my labmates, Drs. (or soon-to-be Drs.) Reyhaneh Jabbarvand, Navid Salehnamadi, Negar Ghorbani, Mahmoud Hammad, Alireza Sadeghi, Forough Mehralian, and Abdulaziz Alshayban. Their support spans far beyond this dissertation. I also want to thank Professor Joshua Garcia, Professor James Jones, Professor Crista Lopes, Professor David Redmiles, and Professor Sameer Singh for guiding me through this significant milestone in my career. Lastly, thanks to my parents and family for supporting me to live the life I wanted.

I appreciate the Ministry of Education, Taiwan and the School of ICS at UCI for supporting my research.

VITA

Jun-Wei Lin

EDUCATION

Doctor of Philosophy in Software Engineering	**2021**
University of California, Irvine	*Irvine, California*
Master of Science in Computer Science	**2008**
National Tsing Hua University	*Hsinchu, Taiwan*
Bachelor of Science in Computer Science	**2006**
National Tsing Hua University	*Hsinchu, Taiwan*

RESEARCH EXPERIENCE

Graduate Student Researcher	**2016–2021**
University of California, Irvine	*Irvine, California*
Research Intern	**2014–2015**
National Agricultural Library	*Beltsville, Maryland*

TEACHING EXPERIENCE

Teaching Assistant	**2019–2020**
University of California, Irvine	*Irvine, California*

REFEREED JOURNAL PUBLICATIONS

The i5k Workspace@NAL—enabling genomic data access, visualization and curation of arthropod genomes **2015**
Nucleic Acids Research

Analysis of Test Suite Reduction with Enhanced Tie-Breaking Techniques **2009**
Information and Software Technology

REFEREED CONFERENCE PUBLICATIONS

Latte: Use-Case and Assistive-Service Driven Automated Accessibility Testing Framework for Android **May 2021**
2021 ACM SIGCHI Conference on Human Factors in Computing Systems (CHI)

Test Automation in Open-Source Android Apps: A Large-Scale Empirical Study **Sept 2020**
35th International Conference on Automated Software Engineering (ASE)

Test Transfer Across Mobile Apps Through Semantic Mapping **Nov 2019**
34th International Conference on Automated Software Engineering (ASE)

Search-Based Energy Testing of Android **May 2019**
41st International Conference on Software Engineering (ICSE)

Nemo: Multi-Criteria Test-Suite Minimization with Integer Nonlinear Programming **May 2018**
40th International Conference on Software Engineering (ICSE)

Using Semantic Similarity in Crawling-Based Web Application Testing **Mar 2017**
10th IEEE International Conference on Software Testing, Veri cation and Validation (ICST)

Automated Testing of Web Applications with Text Input **Dec 2015**
2015 IEEE International Conference on Progress in Informatics and Computing

Test Suite Reduction Analysis with Enhanced Tie-Breaking Techniques **Sept 2008**
4th IEEE International Conference on Management of Innovation and Technology

ABSTRACT OF THE DISSERTATION

Advancing Automated Software Testing Through Test Reuse

By

Jun-Wei Lin

Doctor of Philosophy in Software Engineering

University of California, Irvine, 2021

Professor Sam Malek, Chair

Testing is an indispensable phase of software development life cycle. It is the primary way through which the quality of software is improved. To reduce the cost of manual testing, many automated test input generation techniques have been proposed. Despite all these efforts, current automated techniques still suffer from a set of common limitations that undermines their viability. On the other hand, while rarely discussed or leveraged in the context of test generation, the concept of test reuse has great potential for addressing these limitations. In other words, automated test generation can significantly benefit from reusing and extracting human knowledge from existing test suites.

This dissertation proposes to advance automated software testing through enabling test reuse (1) across similar applications within a platform; (2) across different platforms for the same application; and (3) within an application. To show the feasibility of these ideas, this thesis particularly presents three automated tools, namely CraftDroid, TransDroid, and Route, to (1) transfer GUI tests across similar Android apps; (2) transfer GUI tests for an app from web to its Android counterpart; and (3) augment existing test suites of an Android app to verify the same features but with alternative execution paths. These tools are backed by program analysis and natural language processing techniques to find the semantic mapping between GUI events across apps and platforms. The ability to reuse context-aware text inputs and oracles in the original tests distinguishes this research from prior work.

All conducted experiments on real-world subject apps corroborate the effectiveness and efficiency of the proposed approaches, and their ability to supply automated software testing with test reuse.

Chapter 1

Introduction

As known to everyone, software should be adequately tested before release. What not well-known, however, is that software testing is notoriously tedious and labor-intensive, requiring up to 50% of software development costs [36]. As a result, while in textbook software testing is an indispensable phase of the software development life cycle, in reality it is usually conducted in a casual and ad hoc way, if performed at all [43]. Nevertheless, skipping or overlooking software testing for saving the cost may pay a higher price later. A recent example is the glitch in a mobile app used to transmit results from the Iowa presidential caucuses, which could have been avoided with basic due diligence, including field and user tests [28, 29].

To reduce the cost of software testing, a great deal of research effort has been devoted to automated input generation (AIG) techniques. In mobile app testing, for instance, many such techniques have been proposed in the literature over the past years [1, 2, 35, 37, 48, 70, 82, 75, 89, 94, 95, 97, 103, 128, 130, 131, 123, 92, 59]. Despite all these efforts, current AIG techniques are still far from widespread adoption in practice. For example, several studies indicate that the vast majority of the mobile app testing is still performed manually

[77, 81, 90]. There are three main limitations that undermine the viability of such techniques:

(1) Inability to select valid (text) inputs. Generating valid text inputs is critical to thoroughly test software. For instance, we need proper city and street names for testing navigation software, and correct URLs for testing a browser. Without valid values for such inputs, the generated tests tend to be shallow, failing to reach many interesting states in the app under test (AUT). Case in point, among input generation techniques for apps, random exploration strategy performs just as well—and in most cases better—than more sophisticated approaches, as they are all limited by their inability to produce valid text inputs [49]. Although asking users to manually generate these values defeats the very purpose of automated testing, finding a way to reuse manually-generated values is likely to help test generation considerably.

(2) Inability to create test oracles. Despite a few attempts at automatic generation of test oracles (e.g., [130, 74]), most existing test generation techniques only focus on generating input events alone. The generated tests are therefore unable to identify failures that do not result in a crash. This limitation dramatically reduces the usefulness of automatically generated tests. Also in this case, asking a human to manually create an oracle for each generated test would be impractical. Moreover, creating an oracle for tests whose purpose is unclear would make this task even harder.

(3) Inability to generate meaningful tests. The majority of the existing automated testing techniques aim to either maximize code coverage or reveal as many crashes as possible. This is one of the reasons why the tests generated by such techniques tend to be feature-irrelevant and do not usually reflect the common usage scenarios of an AUT [89, 126]. In addition to the obvious problem that these tests fail to cover relevant use cases, they also make debugging more complex and ineffective. Developers have a hard time debugging tests whose purpose is unclear and that are expressed in terms of low-level commands [90].

While rarely discussed or leveraged in the context of test generation, the concept of test reuse has excellent potential to address the above limitations. First, human knowledge can be retrieved from existing tests and reused to generate context-aware test inputs. Second, since test reuse is only possible among applications provisioning similar functionality and sharing some of their features, the generated tests are inherently feature-relevant and expressive. Finally, by reusing not only test inputs but also assertions or oracles, the generated tests can verify the behavior of the AUT.

This research is also motivated by several key insights. First, modern software systems are intrinsically redundant as a by-product of modern modular software design [47]. This redundancy is observed in software systems at every level of granularity, from source code to component to functionality. For example, a recent study [93] has shown that around 70% of the code on GitHub consists of clones of previously created files. Considering Android apps, for instance, within each category of apps in Google Play Store, there are many highly overlapping apps that share much of their functionality even when providing different user interfaces. As a result, test cases for one application may be reusable for other similar applications.

Second, while the back-end implementation can be different, e.g., with diverse programming languages, frameworks, and platforms, GUI interfaces for the same functionality are usually semantically similar, even if they belong to different software and have different looks and styles. By semantic similarity, we mean the conceptual relation between the textual information, e.g., the text or variable names, which can be retrieved from actionable GUI controls such as buttons and input fields. For example, a button for checkout in different shopping apps can appear as “Confirm and Pay” or “Place Order”, i.e., two syntactically different but semantically similar names. If the mapping of such GUI controls can be built, it is possible to migrate GUI events from one application to another.

Finally, the fact that there are existing test cases in open-source software systems also benefits test reuse. For instance, our empirical study [87] shows that there are hundreds of open-source Android apps with thousands of GUI tests that are readily available and potentially reusable in the wild. Consequently, as code reuse is already a common practice in modern software engineering, test reuse has enormous potential to be a practical solution for everyday use.

1.1 Dissertation Overview

This dissertation proposes a four-pronged approach to advance automated software testing through test reuse.

1. *Intra-platform Test Transfer* discusses the idea and challenges to automatically reuse and migrate existing test suites, including the oracles (assertions), across similar applications within a platform. This thesis addresses a couple of critical challenges and demonstrates the feasibility of this idea by presenting CraftDroid, an automated tool that transfers GUI tests across similar Android apps.

2. *Inter-platform Test Transfer* further presents the need and potential to perform such test migration across platform, as well as the new challenges that prior work in intra-platform transfer has not addressed. The idea of inter-platform test transfer is realized as TransDroid, an automated tool that transfers GUI tests from a web app to its Android counterpart.

3. *Feature-based Test Augmentation* applies test reuse within an application and in the context of test augmentation or amplification. This idea is realized as an automated tool, Route, that augments existing test suites of an Android app to verify the same features but with alternative execution paths. The ability to reuse context-aware text inputs and oracles in the original tests distinguishes the tool from prior work.

4. *A Large-Scale Empirical Study on Test Automation in Open-Source Android Apps* addresses the limitations of previous studies in terms of both the scale of study and the quality of dataset. The findings in this study not only help locate potentially reusable mobile tests, but also shed light on the current practices and future research directions pertaining to test automation for mobile app development.

1.2 Dissertation Structure

The rest of this dissertation is organized as follows. Chapter 2 provides an overview of the prior related research and identifies the position of this work in the research landscape. Chapter 3 presents the research problem and the scope of this thesis. Chapter 4 presents the challenges and techniques for intra-platform test transfer. Chapter 5 shows the proposed framework for inter-platform test transfer. Chapter 6 introduces the automated tool for feature-based test augmentation. Chapter 7 reports the empirical study on test automation in open-source Android apps. Finally, Chapter 8 concludes the dissertation with future work.

The research presented in this dissertation has been published in or submitted to the following venues:

- Jun-Wei Lin, Reyhaneh Jabbarvand, and Sam Malek, Test Transfer Across Mobile Apps Through Semantic Mapping, *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, San Diego, USA, September 2019.
- Jun-Wei Lin, Navid Salehnamadi, and Sam Malek, Test Automation in Open-Source Android Apps: A Large-Scale Empirical Study, submitted to *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Virtual Event, Australia, September 2020

- Jun-Wei Lin and Sam Malek, GUI Test Transfer from Web to Android, submitted to *2022 IEEE International Conference on Software Testing, Verification and Validation (ICST)*

- Jun-Wei Lin, Navid Salehnamadi, and Sam Malek, ROUTE: Roads Not Taken in UI Testing, submitted to *2022 44th IEEE/ACM International Conference on Software Engineering (ICSE)*

In addition, the following publications are not included in the dissertation but are related:

- Jun-Wei Lin, Reyhaneh Jabbarvand, Joshua Garcia, and Sam Malek, Nemo: Multi-criteria Test-Suite Minimization with Integer Nonlinear Programming, *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, Gothenburg, May 2018.

- Reyhaneh Jabbarvand, Jun-Wei Lin, and Sam Malek, Search-Based Energy Testing of Android, *41st International Conference of Software Engineering (ICSE)*, Montreal, Canada, May 2019.

- Navid Salehnamadi, Abdulaziz Alshayban, Jun-Wei Lin, Iftekhar Ahmed, Stacy Branham, and Sam Malek, Latte: Use-Case and Assistive-Service Driven Automated Accessibility Testing Framework for Android, *2021 ACM SIGCHI Conference on Human Factors in Computing Systems (CHI)*, Yokohama, Japan, May 2021.

Chapter 2

Related Work and Research Gap

This chapter reviews the related work that lays the foundation for the proposed research. Particularly, it discusses previous studies that apply automated test reuse in three different contexts and how the proposed techniques are different from them. It then further identifies the research gap and positions the proposed research in the body of literature.

2.1 Related Work

This section discusses prior work related to automated test reuse. It first reviews the research that motivates and justifies automated test reuse, followed by previous studies that apply automated test reuse in three different contexts: intra-platform test transfer, cross-platform testing, and test augmentation. Finally, prior empirical studies on mobile app testing are examined.

2.1.1 Common Functionalities Across GUI-Based Apps

Modern software systems are intrinsically redundant as a by-product of modern modular software design [47]. This redundancy is observed in software systems at every level of granularity, from source code to component to functionality. For example, a recent study [93] indicated that around 70% of the code on GitHub consists of clones of previously created files. As for such redundancy at the functionality level, several prior works have discussed common functionalities across desktop software [98], application-agnostic features across mobile apps [130], and common GUI patterns used in web app testing [59, 106]. Augusto [98] studied common functionalities such as authentication and saving a file in desktop software, and proposed an automated technique to generate GUI tests for them with pre-defined GUI structures and formal pre/post conditions. Zaeem et al. [130] introduced several application-agnostic UI interactions, which can serve as oracles for mobile testing. Ermuth and Pradel [59] proposed that sequences of low-level UI events, which correspond to high level logical steps that can be inferred from test traces and also be further used for test generation. Moreira et al. [106] developed a domain-specific language to model guidelines or recurring solutions for common GUI design problems, and leveraged the model to automate GUI testing. This dissertation shares the same insight as the above works that the existence of commonality across GUI-based apps can be exploited. However, unlike them, proposed approaches in this study generate feature-based tests by directly reusing existing tests from similar applications.

2.1.2 Intra-Platform Test Transfer

In recent years, researchers have proposed various approaches for transferring or reusing tests on different platforms. Rau et al. [114] proposed a technique for mapping of GUI widgets among web applications. Behrang and Orso [41] proposed an approach to transfer test cases by mapping the GUI widgets to support assessment of mobile app coding assignments. Hu et al. [73] presented a framework that leverages machine learning to synthesize reusable UI tests for mobile apps. Qin et al. [112] recently proposed to migrate GUI events for the different instances of the same app running on iOS and Android. While these studies discussed GUI element mapping and shed light on test reuse across applications within a platform, they were not able to transfer test oracles or assertions. Unlike all prior work, CraftDroid is designed to fully migrate tests across applications, and generate tests with oracles that verify the features of the target application.

It is noteworthy that, concurrent to the development of CraftDroid, Behrang and Orso proposed AppTestMigrator [42] to migrate GUI tests, including oracles, for mobile apps with similar functionality. While both works adopt similar techniques at a high level, such as using Word2Vec models and combining static and dynamic analyses, CraftDroid is different from AppTestMigrator in terms of several algorithmic details, such as the ways it leverages the statically extracted model of app and computes similarity between GUI widgets. In a recent study evaluating the techniques for usage-based test reuse across Android apps [133], CraftDroid slightly outperformed AppTestMigrator in terms of the considered effectiveness metrics.

2.1.3 Cross-Platform Testing

A significant number of previous studies have focused on cross-platform testing [100, 52, 50, 51, 62, 117, 76]. Developed concurrently by different research groups, CrossT [100] and WebDiff [52] addressed the problem of cross-browser inconsistencies (XBIs) in web apps, i.e., the same web application behaves differently on different browsers. CrossCheck [50] combined the approaches in CrossT and WebDiff, and leveraged machine learning techniques such as decision tree to improve the accuracy of the reported XBIs. Later, based on an extensive study of XBIs in real-world web apps, X-PERT [51] proposed a framework applying different techniques for different types of XBIs to increase the effectiveness. Regarding the similar presentation issues in Android apps caused by device fragmentation, DiffDroid [62] combined input generation and differential testing to find cross-device inconsistencies. On the other hand, FMAP [117] analyzed the client-server communication of the desktop version and mobile version of a web app to identify missing features in either version. Similar to FMAP, CheckCAMP [76] proposed to identify missing functionality in either the iOS or Android version of the same app. All of these studies in cross-platform testing focus on verifying the assumption that an application's behavior should be consistent on different platforms, whereas TransDroid leverages this assumption to migrates feature-based tests across platforms.

2.1.4 Test Augmentation

Test augmentation techniques [111, 102, 129, 71, 125, 46, 65, 39, 121, 80, 132, 33] create new tests from existing ones to achieve a given engineering goal, such as improving coverage according to a criterion. Pezze et al. [111] proposed to leverage existing unit tests to construct more complex tests that focus on class interactions to reveal more faults. Yoo and Harman [129] introduced a search-based technique that can generate additional test data from existing test data to improve the input space coverage. Harder et al. [71] presented a test augmentation technique based on *operational abstraction*, which is a formal specification of program's runtime behavior that can be dynamically mined. A test suite can be augmented by adding test cases until the operational abstraction stops changing. Tillmann and Schulte [125] proposed to use symbolic execution and constraint solving to help increase code coverage by finding relevant variations of existing unit tests. Similarly, Bloem et al [46] used symbolic execution and model checking techniques to alter path conditions of existing tests and generate new tests that enter uncovered features of the program. Starting from concrete unit tests, Fraser and Zeller [65] presented an approach to generate parameterized unit tests containing symbolic pre- and post-conditions to achieve higher code coverage. To improve the mutation score of an existing test suite, Baudry et al. [39] introduced a genetic algorithm to guides the search for test cases that kill more mutants. Focusing on the context of regression testing, the work by Santelices et al [121] adopted dependency analysis and symbolic execution to synthesize new tests with respect to the code changes not covered by existing tests. Another work considering test-suite augmentation for code changes by Kim et al. [80] leverages different test generation algorithms dynamically, since different algorithms have different strengths. Finally, Zhang and Elbaum [132] developed a solution to amplify a test suite for finding bugs in exception handling code.

Route is different from the prior work because it is for GUI tests, while all of the above augmentation techniques are for unit tests. Furthermore, Route aims to generate tests that verify the same functionality as the original tests, which is not the focus of prior work. Finally, unlike Route, none of the above-mentioned techniques target Android apps.

Route is more related to Thor proposed by Adamsen et al. [33]. Thor takes existing UI tests for Android apps and injects neutral event sequences to see if the original assertions still pass. Neutral event sequences are a series of operations that are not expected to affect the outcome of the injected test cases, such as rotating the phone or turning the screen off and on. Unlike Route, Thor is not able to find alternative tests for verifying the same functionality, because its goal is to simply expose the AUT to adverse conditions.

Another work related to Route is Testilizer for *web applications* proposed by Fard et al. [102]. Testilizer first infers a state flow graph from an existing web test, dynamically explores the graph, and then generates new tests from the updated graph. The goal of Testilizer is to explore new states and apply new and generic assertions learned from existing ones. In other words, the augmentation by Testilizer is not feature-based. It is also not applicable to Android apps.

2.1.5 Empirical studies on Mobile App Testing

Previously, researchers have investigated how mobile test automation is practically adopted [81, 90, 54, 55, 77, 91, 110]. Kochhar et al. [81] analyzed over 600 Android apps on F-Droid to check the presence of test cases and computed the code coverage. They also conducted surveys to understand the usage of automated testing tools and the challenges faced by developers while testing. Cruz et al. [55] analyzed 1,000 Android apps on F-Droid to check their usage of automated testing frameworks and continuous integration tools. They also found that projects using automated testing have more contributors and commits on GitHub. Recently, Fabiano et al. [110] examined 1,780 Android apps on F-Droid to investigate the prominence of tests developed for the apps, as well as other quality metrics of the tests such as test smells, code coverage, and assertion density. Our work is different from theirs in terms of the scale and data source, as we analyzed over 12,000 apps across 16 app markets.

In addition, Coppola et al. [54] studied more than 15,000 apps on GitHub to examine the diffusion, evolution, and modification causes of UI tests in open-source Android apps. While their work is highly related to ours, the key difference is that we focused only on non-trivial apps as they did not factor out toy apps and forks of real apps from their dataset. The inclusion criterion for non-trivial apps makes the dataset considered in our study very different from theirs. For example, among the list of 1,042 repositories with tests released by the authors, only 42 (4%) of them are included in our study.

On the other hand, to understand the main challenges that developers face while building mobile apps, Joorabchi et al. [77] conducted a qualitative study with 12 mobile developers from 9 companies, followed by a survey with 188 respondents. Linares-Vásquez et al. [90] also analyzed responses from 102 open-source Android app developers to understand their practices and preferences regarding Android app testing. Unlike our work, these papers did not analyze open-source data in the wild and merely relied on interviews and survey responses.

Finally, Linares-Vásquez et al. [91] reviewed the frameworks, tools, and services for automated mobile testing, and their limitations. From a survey, they identified several key challenges that should be addressed in the near future by the researchers in the area of mobile test automation. Nevertheless, their work did not include any source code analysis or developer survey.

2.2 Research Gap

This section identifies the research gap in the related literature as follows:

- **Ability to leverage existing oracles for test reuse.** Oracles or assertions in existing tests are the most critical part for automated test reuse to be a practical solution. Without migrating them, the generated tests are not able to verify whether the outcome is correct, which significantly influences the usefulness of prior test generation tools.
- **Inter-platform test transfer.** Many organizations provide their software services on multiple platforms. For example, 80% of the top 50 most visited websites in the United States [3] provide native mobile apps for their users. However, at the state-of-the-practice, despite substantial overlap among several versions of an app provisioned by an organization and intended for execution on different platforms, developers have to manually write separate sets of tests for each version of app. Current techniques for intra-platform test transfer are not applicable when such a tranfer is across platforms.
- **Feature-based UI test augmentation**. Core features (functionalities) of an app can often be accessed and invoked in several ways, i.e., through alternative sequences

of user-interface (UI) interactions. Given the manual effort of writing tests, developers often only consider the typical way of invoking features when creating the tests (i.e., the "sunny day scenario"). However, the alternative ways of invoking a feature are as likely to be faulty. These faults would go undetected without proper tests. Current test augmentation techniques are not able to find additional high-quality UI tests, consisting of both inputs and assertions, that verify the same feature as the original test in alternative ways.

- **Large-scale empirical study considering non-trivial Android apps under real-world contexts.** To understand the test automation culture prevalent among mobile app developers, researchers have investigated the extent to which test automation is adopted in practice. However, those studies are limited in terms of both the scale and the quality of the dataset. Specifically, most prior works have only considered hundreds of apps from a single source. Moreover, previous studies have failed to exclude trivial apps (e.g., class assignments and tutorials) and dummy tests (i.e., the placeholder tests automatically generated by development tools), which might severely affect their conclusion.

Chapter 3

Research Problem

3.1 Problem Statement

Usage-based user-interface (UI) testing or end-to-end testing is a primary way to examine the functionality and usability for GUI-based software. To reduce the cost of manual UI testing, many automated input generation (AIG) techniques have been proposed in the literature over the past years. Considering mobile applications as an example, we see a great deal research has been conducted on such techniques [1, 2, 35, 37, 48, 70, 82, 75, 89, 94, 95, 97, 103, 128, 130, 131, 123, 92, 59]. In general, these techniques follow different exploration strategies, such as random, model-based, stochastic, or search-based, to generate inputs in order to achieve a pre-defined testing goal, e.g., maximizing code coverage or finding more crashes. Despite all these efforts, current AIG techniques still suffer from a set of common limitations that undermines their viability (recall Chapter 1). In short, the problem statement of this thesis can be summarized as follows:

Current AIG techniques for UI testing are limited by inability to (1) select valid text inputs, (2) create test oracles, and (3) generate meaningful tests

3.2 Thesis Statement

Motivated by several key insights, the limitations of current AIG techniques can be addressed by test reuse (recall Chapter 1). Intuitively, test reuse is tantamount to migrating human knowledge and testing effort across applications and platforms. When successful, this process would result in tests that (1) use valid inputs, (2) contain suitable oracles, and (3) are meaningful and representative, thus directly tackling the main issues described in the previous section. In this context,

The knowledge embedded in existing software tests can be automatically reused for advancing automated software testing, such that a generated test comprises valid text inputs and proper test oracles, and reflects meaningful usages of the software under test.

3.3 Research Hypotheses

This dissertation proposes three directions applying test reuse to automated software testing. The first is test transfer across similar applications within a platform. That is, existing test suites for one app, including the test inputs and oracles, can be reused for testing other apps with similar functionalities. There are two main challenges in such transfer. The first is that the mapping of test steps between two apps with the same functionality may not be one-to-one. Second, the mapping of GUI controls is challenging, especially if they are syntactically different but semantically similar (detailed in Chapter 4). Nevertheless, these challenges can be resolved through program analysis and natural language processing techniques. Given that GUI tests typically consist of events that perform actions on GUI controls, such transfer is possible through appropriate mapping of GUI controls in different apps.

Hypothesis 1: UI tests, including test inputs and oracles, can be transferred across similar applications within a platform by developing an automated approach that maps the actionable GUI controls with high precision and recall.

To demonstrate the feasibility of this hypothesis, I will use Android as the platform and develop an automated tool, namely CraftDroid, that is able to transfer GUI tests across similar Android apps. The reason for choosing Android is that mobile applications are among the most used kind of end-user software today, and Android has the largest share of the mobile market at the moment.

The second application of test reuse in automated GUI testing is test transfer across different platforms for the same application. This idea is motivated by a fact that many popular software services can be accessed via multiple platforms (e.g., websites and native mobile apps). Therefore, such inter-platform transfer would be particularly useful in the common case of companies that develop apps with the same functionality for different platforms, forcing the developers to write analogous, yet separate sets of tests for each platform.

Inter-platform test transfer poses new challenges that intra-platform techniques have not addressed: incompatible actions and unclear widget context (detailed in Chapter 5). First, while GUI-based apps share certain common actions such as *click* and *text input*, different platforms usually provide additional unique actions to optimize the user experience. As a result, if the source actions are platform-specific and not supported on the target platform, current techniques are not able to finish the transfer. Secondly, in prior work, the search and mapping of GUI widgets between the source and target apps merely relies on widget context such as type information. However, such context may be missing or ambiguous when the transfer crosses platform boundaries. Therefore, approaches for inter-platform transfer need to tackle these new challenges.

Hypothesis 2: UI tests for shared features of an application can be transferred across platforms by devising an automated approach that maps the actionable GUI controls with high precision and recall.

To show the feasibility of this hypothesis, I will present an automated tool, namely TransDroid, that can transfer GUI tests from a web app to its Android counterpart. The reason for this implementation choice is the fact that the Internet era precedes the smartphone era, and there are a large number of organizations developing their web app prior to their mobile app. As a result, we believe many organizations may benefit from TransDroid, allowing the tests created for their web app to be reused for their mobile app.

Finally, the third direction applying test reuse to automated UI testing is feature-based test augmentation. That is, existing test suites can be reused to examine the same functionalities of the application with alternative execution paths. This idea is based on two observations. First, when writing automated GUI tests, developers usually consider merely the "sunny day scenario" or "happy path", in which the software is used in a typical way. However, real users may not follow the particular way imagined by the developer. Second, essential features of an app can usually be accessed in multiple ways, i.e., with different use-case scenarios. The proposed augmentation can be achieved by obtaining a UI transition model of the application and altering the original execution paths conservatively (detailed in Chapter 6).

Hypothesis 3: Existing UI test suites can be automatically augmented to examine the same features in alternative ways and improve the fault detection effectiveness.

The feasibility of this hypothesis will be demonstrated by an automated tool, namely Route, that is able to perform feature-based test augmentation for Android apps. The tool is different from prior work in test augmentation because it is feature-based, i.e., it aims to generate tests that verify the same functionality as the original tests.

Chapter 4

Intra-Platform Test Transfer

GUI-based testing has been primarily used to examine the functionality and usability of mobile apps. Despite the numerous GUI-based test input generation techniques proposed in the literature, these techniques are still limited by (1) lack of context-aware text inputs; (2) failing to generate expressive tests; and (3) absence of test oracles. To address these limitations, we propose CraftDroid, a framework that leverages information retrieval, along with static and dynamic analysis techniques, to extract the human knowledge from an existing test suite for one app and transfer the test cases and oracles to be used for testing other apps with the similar functionalities. Evaluation of CraftDroid on real-world commercial Android apps corroborates its effectiveness by achieving 73% precision and 90% recall on average for transferring both the GUI events and oracles. In addition, 75% of the attempted transfers successfully generated valid and feature-based tests for popular features among apps in the same category.

4.1 Introduction

GUI testing is the primary way of examining the functionality and usability of mobile apps. To reduce the cost of manual GUI testing, many automated test input generation techniques have been proposed in the literature over the past years [1, 2, 35, 37, 48, 70, 82, 75, 89, 94, 95, 97, 103, 128, 130, 131, 123, 92, 59]. These techniques follow different exploration strategies, such as random, model-based, stochastic, or search-based, for generating inputs in order to achieve a pre-defined testing goal, e.g., maximizing code coverage or finding more crashes. Despite all these efforts to automate the GUI test input generation, several studies indicate that they are not widely adopted in practice and majority of the mobile app's testing is still manual [77, 81, 90]. There are three main reasons that limit the viability of these techniques:

(1) Lack of context-aware text inputs. Most of the state-of-the-art techniques use random input values or rely on the manual configurations for text inputs. However, *contextual* text inputs are critical to thoroughly test majority of the apps, e.g., city names for a navigation app, correct URLs for a browser app, and valid username/password for a mail client app. Without such meaningful inputs, exploration of the App Under Test (AUT) may get stuck at the very beginning and GUI states deep in the testing flow may never been exercised.

(2) Failing to generate expressive tests. Majority of the automated testing techniques aim to maximize the code coverage or reveal as many crashes as possible. The generated tests by such techniques are typically feature-irrelevant or unrepresentative of the canonical usages of apps [89, 126]. This lack of expressiveness makes debugging cumbersome, as such tests do not include the reproduction steps that can be organized by use cases or features [90].

(3) Absence of test oracles. Despite a few efforts for automatic generation of test oracles for mobile apps [130, 74], majority of the existing test generation tools are unable to identify failures other than crashes or run-time exceptions. Without automated test oracles, such tests cannot thoroughly verify correct behavior of the AUT.

To address these limitations, we propose CraftDroid, a framework to reuse an existing test suite for one app to test other similar apps. CraftDroid is inspired by recent work from Behrang and Orso [41] and Rau et al. [114], which provided initial evidence of the feasibility of test transfer for mobile apps and web applications, respectively. Like their works, our proposed technique transfers available test cases corresponding to a specific feature or use-case scenario of one app to other apps with similar functionality. However, unlike their work, CraftDroid is also able to transfer the test oracles, if they exist. To enable context-awareness for text inputs, CraftDroid relies on information retrieval techniques to extract the human knowledge from an existing test suite and reuse it for other apps. Since test transfer is across apps with similar functionalities/features, the generated tests using CraftDroid are inherently feature-relevant and expressive. As CraftDroid not only transfers test inputs, but also oracles (assertions), it is able to thoroughly verify correct behavior of the AUT.

Two insights from the prior literature [40, 98] form the foundation of our work. First, apps within the same category share similar functionalities. For example, shopping apps should implement user registration and authentication to provide personalized services. As another example, web browsers should implement common features such as browsing, adding/removing tabs, or bookmarking URLs, despite different strategies they take for enabling privacy. Second, GUI interfaces for the same functionality are usually semantically similar, even if they belong to different apps with different looks and styles. By semantic similarity, we mean the conceptual relation between the textual information, e.g., the text, adjacent labels, or variable names, which can be retrieved from actionable GUI widgets such as buttons, input fields, or checkboxes. For instance, a button to start the registration process on an app can appear with text "Join", "Sign Up", or "Create Account". Even if the texts are syntactically different, they are semantically related. As another example, a "Confirm and Pay" button on a shopping app for checkout can be a ”Place Order” button on another shopping app.

Extensive evaluation of CraftDroid on real-world commercial and open-source Android apps collected from various categories on Google Play, including popular apps such as Wish, Yelp, and Firefox Focus, confirms effectiveness of the proposed approach. In fact, 75% of the attempted transfers by CraftDroid successfully generated valid and feature-based test cases, with 73% precision and 90% recall on average for the transferred GUI events and oracles. The proposed research in this chapter makes the following contributions:

- A novel technique for transferring both test inputs and oracles across mobile apps through semantic mapping of actionable GUI widgets.
- An implementation of the proposed framework for Android apps, which is publicly available [4].
- Empirical evaluation on real-world apps demonstrating the utility of CraftDroid to successfully transfer tests across mobile apps.

The remainder of this chapter is organized as follows. Section 4.2 introduces a motivating example that is used to describe our research. Section 4.3 provides an overview of our framework and Sections 4.4-4.6 describe the details of the proposed technique. Section 4.7 presents the evaluation results. The chapter concludes with a discussion of the related research and avenues for future work.

4.2 Motivating Example

To illustrate how CraftDroid works, consider Rainbow Shops [5], a shopping app for women clothing, and Yelp [6], a local-search app for services and restaurants. Figures 4.2

and 4.3 show the registration process on Rainbow Shops and Yelp, respectively. To register a new account on Rainbow Shops, user starts by clicking on the "Join" button (Figure 4.2-a), which directs the user to a registration form (Figure 4.2-b). By filling the required fields of registration form and clicking on the "Create Account" button, Rainbow Shops creates an account for the user and moves to the profile page (Figure 4.2-c), which shows information about user, such as her username, i.e., Sealbot.

To initiate the registration process on Yelp, the testing flow starts by clicking on the profile tab, denoted by "Me" in Figure 4.3-a. Then, the user should navigate through several screens to provide required registration information (Figures 4.3-d to 4.3-e). Finally, by clicking on the "Sign Up" button (Figure 4.3-f), the registration process is complete and Yelp moves to the profile page, where user can see her username, i.e., Sealbot (Figure 4.3-g).

While the overall registration process in these two apps follows the same steps—clicking on a button to start registration, filling the registration form, and submitting information—a direct copy of the test steps from Rainbow Shops to Yelp is not possible due to the following reasons: (1) The mapping of test steps between the two apps is not one-to-one. For example, to reach the registration form, Rainbow Shops requires only one click (Figure 4.2-a), while it takes three clicks in Yelp to reach the registration form (Figures 4.3-a, 4.3-b, and 4.3-c). As another example, a user provides personal information using two forms in Yelp compared to the one form in Rainbow Shops. (2) The mapping of GUI widgets is challenging, especially if they are syntactically different but semantically similar. For example, the clicked buttons in these two test flows are different in terms of their label, i.e., "Join" in Rainbow Shops and "Sign Up" in Yelp.

Despite these challenges, CraftDroid is able to transfer a test case that verifies the registration process in Rainbow Shops to Yelp by semantically mapping their GUI widgets. In the following sections, we describe the details of how CraftDroid identifies the matches and transfers GUI/oracle events from Rainbow Shops to Yelp.

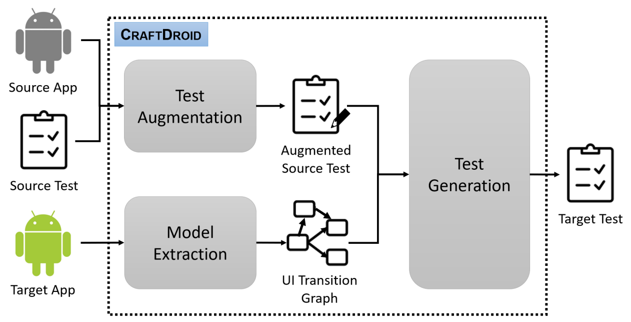


Figure 4.1: Overview of CraftDroid

4.3 Approach Overview

Figure 4.1 provides an overview of CraftDroid consisting of three major components: (1) *Test Augmentation* component that augments test cases available for an existing app, i.e., *source app*, with the information extracted from its GUI widgets that are exercised during test execution, (2) *Model Extraction* component that uses program analysis techniques to retrieve the GUI widgets and identify transitions between Activity components of a target app, and (3) *Test Generation* component that leverages Natural Language Processing (NLP) techniques to compute similarity between GUI widgets of the source and target apps to transfer tests.

More specifically, CraftDroid takes an existing mobile app and its test case as input. It then instruments, executes, and augments the source test with textual information retrieved from the GUI widgets it exercised during its execution. Afterwards, CraftDroid statically analyzes the target app to extract its *UI Transition Graph (UITG)*. Finally, CraftDroid uses UITG of the target app to search for widgets that are similar to those found in the source app to generate a new test. It leverages NLP techniques, such as word embedding, to compute the similarity between GUI widgets in the source and target apps. Regarding the transfer of oracle, CraftDroid is able to deal with several types of oracles that are commonly used in practice, including negative ones such as nonexistence check of text. We will describe these three components in more detail in the following sections.

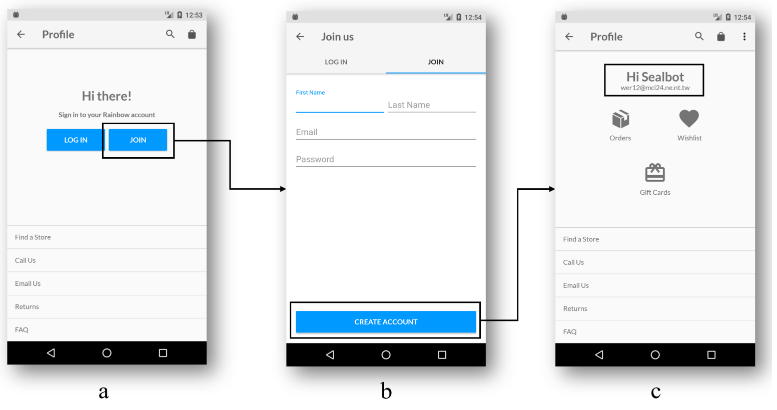


Figure 4.2: Excerpted registration process on Rainbow Shops

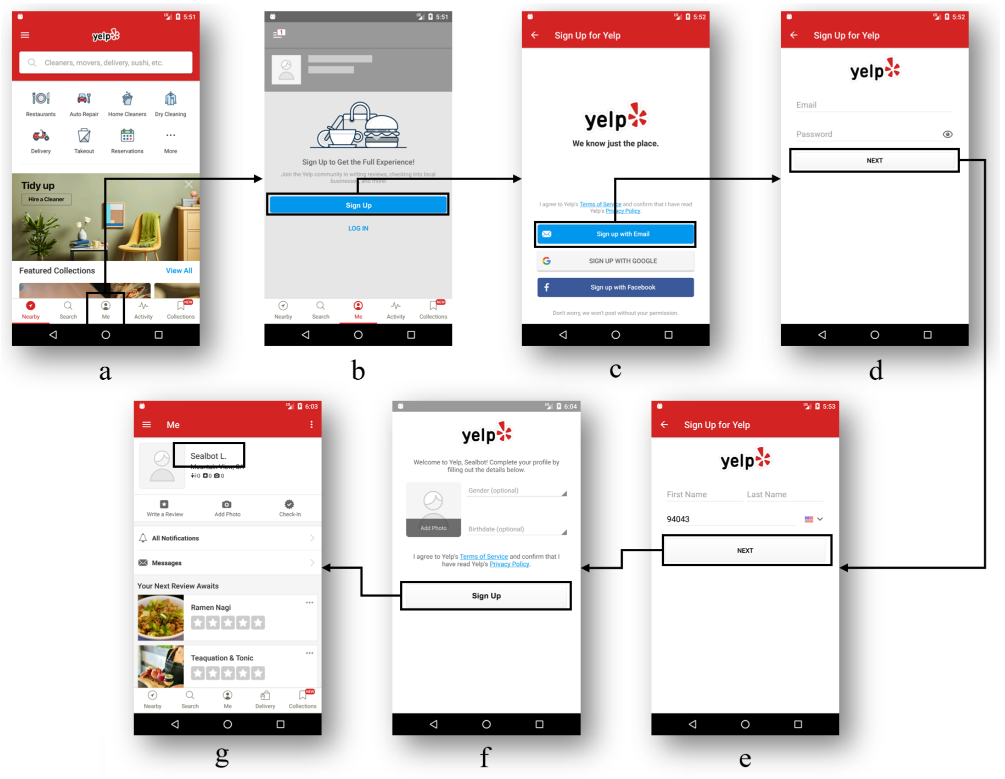


Figure 4.3: Excerpted registration process on Yelp

4.4 Test Augmentation

Algorithm 1 shows how *Test Augmentation* component works. It takes the source app, *srcApp*, with an existing test case, t, as input, and generates an augmented test case t', which contains textual meta-data related to the GUI widgets that are exercised by t. We formulate test case t as a set of *GUI events* $\{(w_1, a_1), (w_2, a_2), ...\}$, where w_i is a GUI widget, e.g., Button, and a_i is the action performed on w_i, e.g., *click*. An action a_i can be a single operation such as *click* or an operation with arguments such as *swipe* that contains starting and ending coordinates. If a test comes with an oracle, CraftDroid identifies it as a special type of event (w_i, a_i), where a_i is an assertion, e.g., *assertEqual*. If the assertion is widget-specific, e.g., existence check of a widget, w_i denotes the widget to be checked. On the other hand, if the assertion is widget-irrelevant, e.g., existence of certain text on the screen, w_i is set to be empty.

Algorithm 1 starts by initializing variables (Line 1) and launching the source app (Line 2). For each GUI or oracle event (w_i, a_i) in t, *Test Augmentation* component dynamically analyzes current screen to retrieve required textual information of w_i, such as the *resource-id*, *text*, and *content-desc*. To that end, it uses adb tool [7] to dump current screen, i.e., an XML file of current UI hierarchy (Line 4), and parses the XML file (Line 5). Algorithm 1 updates w_i with textual information to produce augmented widget w'_i and adds it to the augmented test (Line 6-7). Finally, it executes w_i (Line 8) to move to the next widget.

4.5 Model Extraction

Model Extraction component statically analyzes the target app, *targetApp*, to generate a model called *UI Transition Graph* (UITG). This model represents how Activities/Fragments of an app interact with each other through invoking GUI widgets' event handlers. UITG

Algorithm 1 CraftDroid: Test Augmentation

```
Input:
    A source app $srcApp$;
    A test case $t = \{(w_1, a_1), (w_2, a_2), ...\}$ for $srcApp$
Output:
    An augmented test case $t' = \{(w'_1, a_1), (w'_2, a_2), ...\}$

 1: $t' = \emptyset$;
 2: $launchApp(srcApp)$
 3: for each $(w_i, a_i) \in t$ do
 4:     $screen = dumpCurrentState()$
 5:     $info = getExtraInfo(w_i, screen)$
 6:     $w'_i = augment(w_i, info)$
 7:     $t' = t' \cup (w'_i, a_i)$
 8:     $execute(w_i, a_i)$
 9: end for
10: return $t'$
```

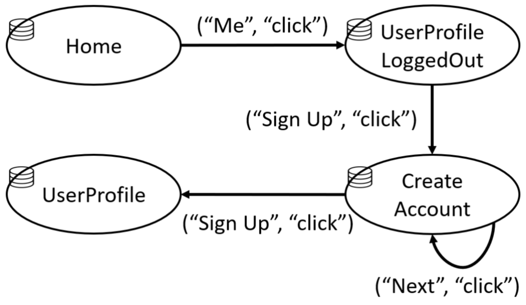


Figure 4.4: Excerpted UI Transition Graph for Yelp

will later be used by *Test Generation* component to search for a match for a given widget of source app in the target app.

At a high level, UITG represents Activity components comprising the target app as nodes and GUI events as transitions among the nodes. Each node of UITG in turn contains a list of widgets that can be rendered directly through the Activity, or indirectly through Fragments comprising the Activity. It is important to consider Fragments, since an Activity may consist of several Fragments, which can be called from different Activities. Figure 4.4 shows a partial *UITG* for Yelp. As demonstrated by this UITG, clicking on the "Me" widget transfers users from the *Home* Activity to the *UserProfileLoggedOut Activity*.

Model Extraction constructs the UITG in two steps:

(1) Extracting Activities, Fragments, and their corresponding GUI widgets. *Model Extraction* parses *Manifest* file of the target app to collect a list of Activity components. For each identified Activity, it then extracts all of the GUI widgets, e.g., Button, EditText, and TextView, that it renders during execution of the app. These widgets are either implemented by the Activity itself or inside Fragments within the Activity.

To extract widget list, *Model Extraction* analyzes XML-based meta-data (*Resource* files)—for statically defined GUI widgets—as well as the source code—for dynamically defined ones. More specifically, to get the list of statically defined GUI widgets, *Model Extraction* first refers to the source code of each Activity/Fragment and looks for specific methods, such as *setContentView()* and *findViewById()*, to identify resource files corresponding to widgets. It then adds all the widgets identified in the resource file to the widget list of the Activity. To get the list of dynamically defined GUI widgets, *Model Extraction* analyzes the source code of Activity/Fragment components to identify initialization of GUI widget elements in them and adds the corresponding widgets to the widget list. During extraction of widgets, *Model Extraction* also retrieves and stores their corresponding textual information.

(2) Identifying transitions between Activities. *Model Extraction* starts from the launcher Activity, which is specifically identified in the *Manifest* file. For each Activity, it analyzes the event handlers of all the GUI widgets in the Activity's widget list, e.g., *onClick()* for a button or *onCheckedChanged()* for a check box. If the event handler of a widget invokes specific methods that result in transition to another Activity (e.g., *startActivity()*) or Fragment (e.g., *beginTransaction()*), *Model Extraction* includes a transition between the two Activity Components. We identify two types of transitions in UITG:

1. *Inter-component transition.* The method call results in a transfer of control between two distinct Activities. For example, when user clicks on the "Me" tab in Figure 4.3-a,

the *onClick()* handler of this widget initiates an Intent message and invokes *startActivity()* method to transfer the control to *UserProfileLoggedOut* Activity.

2. *Intra-component transition.* The method call to a GUI event handler results in a transition back to the same Activity. Such transitions happen when an Activity consists of multiple Fragments and performing an action on one Fragment results in transition to another Fragment within the same Activity. For instance, Figures 4.3-d and 4.3-e represent two Fragments related to the *CreateAccount Activity.* Clicking on the "Next" button on the first Fragment moves the control to the second Fragment. Thereby, this transition causes a loop in the UITG, as shown in Figure 4.4.

After generation of *UITG*, *Model Extraction* component combines the widgets collected for all nodes into an associative array, denoted as *map*. This construct maps all the GUI widgets in *targetApp* to the corresponding Activity/Fragment that can render them during execution of app.

4.6 Test Generation

Algorithm 2 demonstrates how the *Test Generation* component of CraftDroid works. This component takes the *targetApp*, its corresponding *UITG* and widget *map*, and an augmented test for the *srcApp*, t', as input and generates a new test case t_{new} for *targetApp* by transferring the GUI and oracle events of t'.

To that end, it iterates over every GUI or oracle event (w_i, a_i) in t' and collects a list of candidate widgets in *targetApp*, *candidates*, which are ranked based on their similarity to w_i (Line 4, details in Section 4.6.1). For each GUI widget w_n in *candidates*, Algorithm 2 checks to see if it is reachable, and if so, identifies a sequence of events—leading events—that should be executed to reach w_n (Line 6, details in Section 4.6.2).

Algorithm 2 CraftDroid: Test Generation

```
Input:
    $targetApp$,
    $UITG$ of $targetApp$,
    $map$ widgets on each Activity/Frag. in the $targetApp$,
    $t' = \{(w'_1, a_1), (w'_2, a_2), ...\}$ from $srcApp$,
Output:
    $t_{new} = \{(w_{n_1}, a_{n_1}), (w_{n_2}, a_{n_2}), ...\}$ for $targetApp$

1: while $true$ do
2:     $t_{new} = \emptyset$
3:     for each $(w'_i, a_i) \in t'$ do
4:         $candidates = getCandidates(w'_i, map, UITG)$
5:         for each $w_n \in candidates$ do
6:             $leadingEvents =$
                   $getLeadingEvents(w_n, UITG, map, t_{new})$
7:             if $leadingEvents \neq null$ then
8:                 $a_n = generateAction(w'_i, a_i, w_n)$
9:                 $t_{new} = t_{new} \cup leadingEvents \cup (w_n, a_n)$
10:                break
11:            end if
12:        end for
13:    end for
14:    if $\Delta\ fitness(t_{new}) \leq threshold\ or\ timeout$
15:        break
16:    end if
17: end while
18: return $t_{new}$

19: function getLeadingEvents($w_n, UITG, map, t_{new}$)
20:    $execute(t_{new})$
21:    $srcAct = getCurrentActivity()$
22:    $destAct = getActivity(w_n, map)$
23:    $paths = getPaths(srcAct, destAct, UITG)$
24:    $sort(paths)$
25:    for each $path \in paths$ do
26:        $isValid = validate(w_n, path, map)$
27:        if $isValid = true$ then
28:            return $path$
29:        end if
30:    end for
31:    return $null$
32: end function
```

For a reachable candidate w_n, Algorithm 2 identifies the appropriate action a_n (Line 8, details in Section 4.6.3), adds (w_n, a_n) along with the leading events to t_{new} (Line 9), and moves to the next w'_i to find its match (Line 10).

Once all the widgets in t' are checked for a match in *targetApp*, Algorithm 2 checks the termination criteria (Line 14, details in Section 4.6.4). If termination criteria are met, it terminates (Line 15). Otherwise, it repeats the whole process of transfer. The reason for repeating the test generation process is that *Test Generation* component relies on *UITG* to identify reachability of the candidate widgets. Since *UITG* is derived through static analysis, it is an over approximation of the app's runtime behavior. In addition, static analysis is not able to realize dynamically generated contents such as pop-up dialogues or buttons in Android's WebView. To overcome these limitations, CraftDroid executes *targetApp* to determine reachability and updates *UITG* based on runtime information. Thereby, *Test Generation* repeats transfer with an updated *UITG* to increase the chance of successful transfer.

In the remainder of this section, we describe the key components of *Test Generation* in more detail.

4.6.1 Computing Similarity Score

In the `getCandidates` function (Line 4), *Test Generation* considers two factors to compute the similarity between widgets: (1) their corresponding textual information, and (2) their location in *UITG*. More specifically, to determine the similarity of a candidate widget w_n to source widget w'_i, *Test Generation* first computes $score_t$—a measure of how similar are the textual information of w_n to that of w'_i. It then normalizes $score_t$ based on how close w_n is to the current Activity by leveraging *UITG* to compute the final similarity value.

Computing textual similarity score, $score_t$

CraftDroid collects the textual information of a GUI widget from multiple sources, such as widget's attributes, the name of Activity/Fragment that renders it, and its immediate parent and siblings. CraftDroid follows a two step process to measure the textual similarity. It first retrieves raw textual data from different sources and processes them. It then utilizes the processed data to measure the similarity in a weighted scheme among all sources.

Text Processing. *Test Generation* processes the collected textual information by *Test Augmentation* and *Model Extraction* through applying a series of common practices in NLP, including tokenization and stopword removal. The result of this step is a set of word lists for every textual information. For example, textual information for the button *Sign Up* in Figure 4.3-b can have three word lists: (1) ["Sign","Up"] from its label, (2) ["sign", "up", "button"] from its resource-id of *sign_up_button*, and (3) ["user", "profile", "logged", "out"] from its Activity name of *UserProfileLoggedOut*.

Computing Textual Similarity. To determine $score_t$ between two GUI widgets w_n and w'_i, *Test Generation* computes the similarity score for each information source and then calculates a weighted sum of the individual scores. Since the previous step produces a set of word lists for each GUI widget, the problem of determining the textual similarity between two GUI widgets is dual to the problem of computing the similarity score between word lists.

CraftDroid leverages Word2Vec [101]—a model that captures the linguistic contexts of words—to compute the similarity score between two word lists. That is, it first computes the cosine similarity for all possible word pairs in the word lists. Next, it identifies the best match among pairs based on two criteria: (1) the pair has the highest cosine similarity, and (2) every word is only matched once.

For instance, consider the *Create Account* button in Figure 4.2-b and *Sign Up* button in

Figure 4.3-b from the motivating example. The two word lists corresponding to these buttons are [“Create”, “Account”] and [“Sign”, “Up”]. To compute the similarity score between them, the pairwise cosine similarity is calculated as follows:

$$\begin{array}{c} \\ Sign \\ Up \end{array} \begin{array}{c} \begin{array}{cc} Create & Account \end{array} \\ \begin{bmatrix} \mathbf{0.405} & 0.168 \\ 0.201 & \mathbf{0.158} \end{bmatrix} \end{array}$$

In this example, the word pairs that match the mentioned criteria are (”Create”, ”Sign”) and (”Account”, ”Up”) with cosine similarity of 0.405 and 0.158, respectively. Thereby, the final similarity score between these two word lists is calculated as $(0.405+0.158)/2 = 0.282$, which is the score for the *text* of these two buttons. Similarly, the two word lists corresponding to the *resource-id* of these two widgets are [“button”, “sign”, “up”] and [“sign”, “up”, “button”]. The cosine similarity for these lists are as follows:

$$\begin{array}{c|ccc} & button & sign & up \\ \hline sign & 0.117 & \mathbf{1.0} & 0.149 \\ up & 0.048 & 0.149 & \mathbf{1.0} \\ button & \mathbf{1.0} & 0.117 & 0.048 \end{array}$$

Based on these values, the score for *resource-id* is calculated as $(1.0 + 1.0 + 1.0)/3 = 1.0$. If only these two information sources are considered to compute the similarity score, the final textual similarity between these two buttons is $(0.282 + 1.0)/2 = 0.641$.

Computing final similarity score

To compute the final similarity score between w_n and w_i, *Test Generation* normalizes $score_t$ based on the distance of w_n from current screen. *Test Generation* consults *UITG* to get the shortest distance d, i.e., number of GUI events, from the current screen to the Activity to which w_n belongs. It computes the final similarity as follows:

$$similarity(w_n, w'_i) = \begin{cases} score_t, & \text{if } d = 0 \\ \\ \dfrac{score_t}{1 + \log_2 d}, & \text{otherwise} \end{cases}$$

This adjustment assigns a higher priority to candidate GUI widgets that are closer to the current screen. This is because intuitively, the steps or events to test the same functionality should not be significantly different even in different apps. For example, consider the *Join* button from Figure 4.2-a in Rainbow Shops. The most semantically similar widget in Yelp app to this button is the *Sign Up* button, which appears in multiple UIs, e.g., Figures 4.3-b, 4.3-c, and 4.3-f. To identify which one of these buttons is the best match for *Join*, CRAFTDROID starts from the launcher Activity of Yelp, *Home* Activity, and finds the closest node in its *UITG* (Figure 4.4) that contains a *Sign Up* button, *UserProfileLoggedOut* Activity, which is shown in Figure 4.3-b.

4.6.2 Reachability Check

The function `getLeadingEvents` in Algorithm 2 checks the reachability of w_n, a candidate widget in *targetApp* that can be matched to w'_i. If reachable, the function returns the GUI events leading to the Activity holding w_n. To that end, first t_{new}—series of GUI events successfully transferred so far—is executed and the last activity *srcAct* executed by t_{new} is identified (Line 21). The widget *map* is then used to pinpoint the Activity *destAct* that holds w_n (Line 22). Next, all the potential paths in *UITG* from *srcAct* to *destAct* are explored to derive sequences of GUI events—leading events—that execute each path (Line 23).

The identified paths are sorted based on their length (Line 24). This way, shorter paths have a higher chance of being selected, thereby making the length of final transferred test shorter, which is generally desirable for debugging purposes. The function `validate` then verifies whether w_n is reachable by executing actions corresponding to each *path* on *targetApp* (Line 26). The first path that verifies reachability of *destAct* from *srcAct* is returned as output (Lines 27-28). If no path is found or could be verified, `null` is returned (Line 31), indicating that w_n is not reachable.

Finally, it is worth mentioning that in addition to verifying the reachability of each path, function `validate` (1) updates *UITG* by removing invalid paths, i.e., unreachable paths, (2) updates the widget *map* by adding new GUI widgets that are encountered at runtime (i.e., those that are loaded dynamically), and (3) determines the correct screen for asserting negative oracles (details in Section 4.6.3).

4.6.3 Actions for the Transferred GUI and Oracle Events

Once a widget match w_n is found, Algorithm 2 determines the proper action a_n for it to successfully transfer (w_i', a_i) (Line 8). Based on the type of event, i.e., GUI or oracle event, Algorithm 2 identifies a_n as follows:

GUI event. Even when the type of matched GUI widgets in *srcApp* and *targetApp* are the same, their corresponding action might be different. For example, removing an item in a to-do list app can be performed by a swipe, while the same task in another to-do list app might be performed by a long click. To overcome this challenge, CraftDroid considers a series of possible actions for w_n and finds the one that properly works on w_n in *targetApp*. To that end, it first analyzes the source code of *targetApp* to find a specific event listener, such as *onSwiped()* or *setOnLongClickListener()*, registered for the matched widget w_n, and returns a_n as the action corresponding to such an event listener. If no specific action can be

Table 4.1: Types of oracle supported by CraftDroid. (w'_i, a_i): the source oracle event. (w_n, a_n): the transferred target oracle event.

a_i	a_n	Widget-specific?
$assertEqual(VALUE_i, attr(w'_i))$	$assertEqual(VALUE_n, attr(w_n))$	Y
$elementPresense(w'_i)$	$elementPresense(w_n)$	Y
$elementInvisible(w'_i)$	$elementInvisible(w_n)$	Y
$textPresense(STRING)$	$textPresense(STRING)$	N
$textInvisible(STRING)$	$textInvisible(STRING)$	N

identified, it reuses the same action in *srcApp*, i.e., assign $a_n = a_i$.

Oracle event. For oracle events (w'_i, a_i) in *srcApp*, where a_i is an assertion, CraftDroid generates a_n for *targetApp* based on whether a_i is widget-specific, e.g., existence check of a widget, or widget-irrelevant, e.g., existence check of text.

Table 4.1 lists the types of oracle events supported by the current version of CraftDroid. For widget-specific assertions, *Test Generation* modifies the assertion so that it matches the target widget, w_n. For example, when a_i checks if the *resource-id* of w'_i matches a specific value, the generated a_n should also check if the *resource-id* of w_n matches a specific value (First row in Table 4.1). On the other hand, if a_i is widget-irrelevant, it can be directly transferred to *targetApp*.

Transferring negative oracle events, e.g., nonexistence of text, is challenging, as they can make the transferred test pass, regardless of the successful transfer of tests. For example, consider testing the functionality of removing a task from to-do list. To ensure that an item has been successfully deleted, the oracle could be a negative assertion of *textInvisible* to check non-existence of item's text. Suppose that we have a source app that removes a task without confirmation, while target app requires one additional step to get confirmation of removal from user before removing the task. An unsuccessful transfer of test that does not consider user confirmation in target app can still pass, since the negative assertion will be checked at the confirmation step, where the text of item is not visible, yet item is not deleted. Thereby, the main challenge of transferring negative oracles is to identify the correct screen for them to be executed.

A heuristic that allowed us to overcome this challenges is as follows: a negative oracle is likely to be asserted on the proper screen when its negation (i.e., positive oracle) is also asserted on that same screen, albeit with different content displayed on the screen. To find the correct screen for a negative oracle, CraftDroid uses *anchor widget*—an actionable widget that appears in the screen where both a negative oracle and negation of the negative oracle (i.e., positive oracle) should be asserted. The anchor widget serves as a reference to the correct screen. To identify an anchor widget, CraftDroid first negates the assertion of negative oracle and then searches for a screen where that assertion can be verified. Any actionable widget in that screen can be considered as the anchor widget. To that end, CraftDroid analyzes the source test, t'_i, before transfer and determines the negate of negative oracle, if one exists. During test transfer, it examines the negated assertion on all screens and selects an actionable widget in a screen that the negated assertion passes as an anchor widget[1].

In the example of to-do list apps, CraftDroid negates the negative oracle of text non-existence to existence, i.e., checks if the text of an item exists in the current screen. The anchor widget in this example could be an *Add* widget that is used to add items to a list. This is because existence of the text of a to-do item should be checked when that item is being added. Thereby, a widget for adding always exists in the screen that list items exist. Later for transfer of oracle event, CraftDroid leverages *UITG* to first navigate back to the screen, where the *Add* exists, and then transfers the oracle.

4.6.4 Termination Criteria

Algorithm 2 iteratively improves the quality of test transfer through updating *UITG* and the widget *map*. It terminates once the fitness of a transferred test cannot be improved any

[1]CraftDroid uses anchor widget instead of Activity names, since Activity might have multiple Fragments. Thereby, just getting back to the Activity does not guarantee the screen is correct.

further, or a timeout value is reached. The fitness of a transferred test is the average of similarity values (Section 4.6.1) computed for its corresponding events.

4.7 Evaluation

We investigate the following research questions in our experimental evaluation of CraftDroid:

RQ1. How effective is CraftDroid in terms of the number of successful transfers compared to total attempted transfers? What are the precision and recall for attempted GUI and oracle transfers?

RQ2. What are the main reasons yielding transfer failure?

RQ3. How efficient is CraftDroid in terms of the running time to transfer tests from one app to another?

RQ4. What are the factors impacting the efficiency of CraftDroid?

4.7.1 Experimental Setup

We implemented CraftDroid with Python and Java for test cases written using Appium [8], which is an open source and cross-platform testing framework. Existing test cases for the subject apps are written using Appium's Python client and the augmented/generated test cases are stored in JSON format. The *Model Extraction* component is built on top of Soot, a static analysis framework for Java [127]. For our experiments, we used a Nexus 5X Emulators running Android 6.0 (API 23) installed on a Windows laptop with 2.8 GHz Intel Core i7 CPU and 32 GB RAM.

Table 4.2: Subject apps.

Category	App (version)	Source
a1-Browser	a11-Lightning (4.5.1)	F-Droid
	a12-Browser for Android (6.0)	Google Play
	a13-Privacy Browser (2.10)	F-Droid
	a14-FOSS Browser (5.8)	F-Droid
	a15-Firefox Focus (6.0)	Google Play
a2-To Do List	a21-Minimal (1.2)	F-Droid
	a22-Clear List (1.5.6)	F-Droid
	a23-To-Do List (2.1)	F-Droid
	a24-Simply Do (0.9.1)	F-Droid
	a25-Shopping List (0.10.1)	F-Droid
a3-Shopping	a31-Geek (2.3.7)	Google Play
	a32-Wish (4.22.6)	Google Play
	a33-Rainbow Shops (1.2.9)	Google Play
	a34-Etsy (5.6.0)	Google Play
	a35-Yelp (10.21.1)	Google Play
a4-Mail Client	a41-K-9 (5.403)	Google Play
	a42-Email mail box fast mail (1.12.20)	Google Play
	a43-Mail.Ru (7.5.0)	Google Play
	a44-myMail (7.5.0)	Google Play
	a45-Email App for Any Mail (6.6.0)	Google Play
a5-Tip Calculator	a51-Tip Calculator (1.1)	Google Play
	a52-Tip Calc (1.11)	Google Play
	a53-Simple Tip Calculator (1.2)	Google Play
	a54-Tip Calculator Plus (2.0)	Google Play
	a55-Free Tip Calculator (1.0.0.9)	Google Play

Subject apps. We evaluated the proposed technique using both open-source and commercial Android apps. CraftDroid is able to transfer tests for similar functionalities implemented differently on separate apps. Thereby, we performed test transfers among apps within the same category and for each category, identified main functionalities to be tested. To that end, we selected five categories that have large number of apps on Google Play, namely *Browser*, *To-Do List*, *Shopping*, *Mail Client*, and *Tip Calculator*. These five categories are often used in prior research that either studied common functionalities across mobile/web apps [116, 73, 98, 114] or proposed Android GUI testing solutions [49, 97, 104, 95]. Table 4.2 shows the list of 25 subjects and their categories.

For each category, we identified two main functionalities and the corresponding test steps. The test steps for each functionality, which are listed in Table 4.4 include at least one oracle step. The oracle steps are implemented as assertion and wait-until statements.

Table 4.3: Test cases for the proposed functionalities.

Functionality	#Test Cases	Avg# Total Events	Avg# Oracle Events
b11-Access website by URL	5	3.4	1
b12-Back button	5	7.4	3
b21-Add task	5	4	1
b22-Remove task	5	6.8	2
b31-Registration	5	14.2	5
b32-Login with valid credentials	5	9	4
b41-Search email by keywords	5	5	3
b42-Send email with valid data	5	8	3
b51-Calculate total bill with tip	5	3.8	1
b52-Split bill	5	4.8	1
Total	50	6.6	2.4

Test cases. To construct tests suites, we first collected tests for each subject app[2], if there were any, and then augmented the test suites with the test cases corresponding to the steps described in Table 4.4. The number of events for tests among different categories varies from 3 to 19, with an average of 6.6 events, including 2.4 oracle events [3].

Attempted transfers. For each test case validating a functionality of an app, CraftDroid transfers the test case to the other four apps under the same category. Thereby, the number of attempted transfers for each category are 5 (test cases) $\times$ 4 (transfers) = 20, making the total number of attempted transfers for evaluating CraftDroid to be 200. After each transfer, we manually examined the test and its execution to identify *false positive*, *false negative*, and *true positive* cases as follows: *false positive* occurs when the target widget of manual transfer is different from w_n identified by CraftDroid; *false negative* occurs when CraftDroid fails to find a target widget, while manual transfer can; and *true positive* occurs when the target widget from manual transfer matches w_n identified by CraftDroid. Based on these metrics, we measured the *Precision* as the number of generated target events that are correct. Additionally, *Recall* measures how many of the source events are correctly transferred. Our experimental data is publicly available [4].

[2]Test suites for *Geek*, *Wish*, and *Etsy* apps are from [73]

[3]The number of actual GUI and oracle events in the test cases may be more than the number of steps shown in Table 4.4, since Table 4.4 only provides general instructions for testing the functionalities

Table 4.4: Identified main functionalities for subject apps.

Category	Functionality	Test Steps
a1-Browser	b11-Access website by URL	1. Locate the address/search bar
		2. Input a valid URL and press Enter
		3. Specific content about the URL should appear
	b12-Back button	1. Locate the address/search bar
		2. Input valid URL1 and press Enter
		3. Specific content about URL1 should appear
		4. Input valid URL2 and press Enter
		5. Specific content about URL2 should appear
		6. Click the back button
		7. Specific content about URL1 should appear
a2-To Do List	b21-Add task	1. Click the add task button
		2. Fill the task title
		3. Click the add/confirm button
		4. The task title should appear in the task list
	b22-Remove task	1. Add a new task
		2. Click/long-click/swipe the task in the task list to remove the task
		3. Click the confirm button if exists
		4. The task should not appear in the task list
a3-Shopping	b31-Registration	1. Click the register/signup button
		2. Fill out necessary personal data
		3. Click the submit/signup button to confirm registration
		4. Personal data should appear in the profile page
	b32-Login with valid credentials	1. Click the login/signin button
		2. Fill out valid credentials
		3. Click the submit/signin button to login
		4. Personal data should appear in the profile page
a4-Mail Client	b41-Search email by keywords	1. Start the inbox activity
		2. Click the search button
		3. Input keywords for search and press Enter
		4. Specific email related to the keywords should appear
	b42-Send email with valid data	1. Start the inbox activitiy
		2. Click compose button
		3. Input an unique ID for the subject
		4. Input a valid email address for the recipient
		5. Click send button
		6. The unique ID should appear in the inbox
a5-Tip Calculator	b51-Calculate total bill with tip	1. Start the tip calculation activity
		2. Input bill amount and tip percentange
		3. Total amount of bill should appear based on the values in step 2
	b52-Split bill	1. Start the tip calculation activity
		2. Input bill amount and tip percentage
		3. Input number of people
		4. Total amount of bill per person should appear based on the values in step 2 and 3

4.7.2 RQ1: Effectiveness

Table 4.5 demonstrates the effectiveness of CraftDroid in terms of successful transfers for each functionality listed in Table 4.4. These results demonstrate that on average, 74.5% of the attempted transfers by CraftDroid are successful, with an overall 73% precision and 90% recall considering all the transferred GUI and oracle events. Thereby, CraftDroid is substantially effective in identifying correct GUI widgets and successfully transferring tests across mobile apps.

Table 4.5: Effectiveness and Efficiency Evaluation of CraftDroid.

Functionality	GUI Event		Oracle Event		#Successful Transfer	Avg. Transfer Time (sec)
	Precision	Recall	Precision	Recall		
b11	79%	100%	100%	100%	20/20 (100%)	1,144
b12	85%	100%	100%	100%	20/20 (100%)	4,986
b21	78%	100%	85%	100%	17/20 (85%)	1,051
b22	69%	100%	85%	80%	11/20 (55%)	10,611
b31	44%	90%	34%	67%	8/20 (40%)	14,974
b32	53%	82%	56%	61%	10/20 (50%)	8,644
b41	100%	100%	100%	100%	20/20 (100%)	349
b42	85%	80%	89%	89%	14/20 (70%)	2,611
b51	82%	100%	100%	80%	16/20 (80%)	2,581
b52	80%	100%	100%	65%	13/20 (65%)	6,762
Total	70%	94%	79%	85%	149/200 (74.5%)	5,371

The results shown in Table 4.5 also confirm that finding a match for all the widgets in source test is not necessary to successfully transfer a test. As an instance for such cases, consider the functionality *b11*, where its corresponding precision for transferring GUI events is 79%, while it successfully transfers all tests (success rate = 100%). That is, transfer of events from source app to target app in *b11* has been accompanied by false positives, i.e., the target widget is identified incorrectly. However, these false positives are not harmful, since different apps might implement common functionalities in different ways. For example, while one app may require the user to confirm the provided password during registration and before an account is created, this confirmation may not be required in another app, thereby, can be skipped.

While false positive may be acceptable, false negative is not, as it prevents the examination of the desired functionality. In other words, high recall is more important than high precision in test transfer, as false negatives typically have more adverse affect compared to false positives. CraftDroid's high recall of 90% for transferring GUI and oracle events makes it suitable for test transfer.

Another important observation from the results in Table 4.5 is that the success rate varies significantly among different categories of apps, ranging from 100% success rate for the apps under *Browser* and *Mail Client* categories to 40% for Shopping apps. Even within the same category, the success rate varies for different functionalities. In the next research question,

Table 4.6: Pearson correlation coefficient between average test length and effectiveness.

	GUI Event		Oracle Event		#Successful Transfer
	Precision	Recall	Precision	Recall	
Avg. Test Length	-0.74	-0.60	-0.87	-0.51	-0.71

we investigate the attributes that impact the success rate of test transfer.

4.7.3 RQ2: Factors Impacting Effectiveness

To identify the factors that impact effectiveness of a test transfer, we manually investigated all of the attempted transfers, including both successful and failed ones. We identified the following reasons for transfer failure:

Length of test. Intuitively, transfer of a long test is more challenging compared to a shorter one, since more GUI and oracle events should be transferred. Thereby, more false positives and false negatives might be generated. To identify how the length of tests impact effectiveness of a test transfer, we calculated the Pearson correlation coefficient [9] between the average length of tests, i.e., number of total events, and the effectiveness metrics in our experiments. Table 4.6 represents the computed correlation coefficients. These results indicate a strong and negative correlation between the length of tests and success of a test transfer.

Complexity of app. Complexity of subject apps, in terms of their interface and functionality, also impacts the effectiveness of CraftDroid. Some categories of apps have standard or de-facto design guidelines, such as arrangement of GUI widgets to follow, which makes the transfer of test cases across such apps easier. For example, the design guideline for browser apps is to have a simple main screen that only contains a search bar and few actionable GUI widgets. This relatively simple design for the browser apps makes the transfer of the GUI events across them easier, since there are fewer candidate widgets on a screen to be analyzed for proper mapping. On the other hand, apps without uniform design guidelines, such as Shopping apps, are flexible to determine the number of functionalities contained on a screen and the number of required steps for a functionality. This flexibility makes the search for finding correct matches more complicated. As demonstrated by the results in Table 4.5, while CraftDroid successfully transfers all the tests under the Browser category, the success rate of transfer among Shopping apps is not as high as other categories.

4.7.4 RQ3: Efficiency

Table 4.5 shows the average running time of CraftDroid to transfer a test from one app to another when executed sequentially. On average, a test transfer takes less than 1.5 hours, ranging from 6 minutes to 4.2 hours among different functionalities. Performance evaluation of different components of CraftDroid shows that validating reachability of a candidate widget is the most time-consuming part of test transfer. That is, the function `getLeadingEvents` in Algorithm 2 dominates the execution time, as it frequently restarts the target app to validate the potential paths for a candidate widget.

Fortunately, this function can be easily parallelized. Multiple devices or emulators can be used to drastically cut down the execution time by performing the reachability check in parallel. For example, in our experiments, CraftDroid verifies 6.6 paths on average to transfer a source event. As a result, by using 6 emulators, we can speed up the transfer approximately 6 times to reduce the average running time to 15 minutes. We believe this is a reasonable amount of time to produce a feature-based test, consisting of both inputs and oracles.

4.7.5 RQ4: Factors Impacting Efficiency

By analyzing efficiency of test transfer among different apps and functionalities, we identified three factors that impact the efficiency of CraftDroid: (1) length of tests, (2) transfer success, and (3) size of target app. Intuitively, the longer is a test, it takes more time to transfer its events. In fact, the average test length and average transfer time are strongly and positively correlated, as the Pearson correlation coefficient between them is 0.81 in our experiments.

In addition, we observed that unsuccessful transfers take more time compared to successful ones. That is, an unsuccessful transfer often needs to examine and validate more candidate widgets during transfer. In our experiments, the average running time of the 149 successful transfers is $3,577$ seconds, while this number for the remaining 51 unsuccessful transfers is $10,613$ seconds, meaning unsuccessful transfers are $3x$ slower. Finally, the size of *UITG* is positively correlated to the size of app—with correlation coefficient = 0.5. Since CraftDroid heavily relies on the *UITG* to search and validate the correct widget, it requires more time to transfer a test for a larger target app.

4.7.6 Threats to Validity

The major external threat to validity of our results is the generalization to other mobile apps and test cases. To mitigate this threat, we collected 25 commercial and open-source apps from Google Play and F-Droid under various categories. The main internal threat to validity of the proposed approach is the possible mistakes involved in our implementation and experiments. We manually inspected all of our results to increase our confidence in their correctness. The experimental data is also publicly available for external inspection. In terms of the construct validity, CraftDroid assumes that the source test is transferable, i.e., the source and the target apps share similar functionalities, which is not always true.

However, CraftDroid is not designed to generate test cases for every possible or app-specific features. It aims at reducing the manual effort of implementing tests for common or popular functionalities across apps. Our evaluation shows that this assumption does hold for apps under different categories.

4.8 Conclusion and Future Work

In this chapter, we presented CraftDroid, a framework for transferring tests across mobile apps through semantic mapping of actionable GUI widgets. We evaluated CraftDroid using 25 real-world apps from 5 categories. Our experimental results show that 75% of the attempted transfers are successful, with 73% precision and 90% recall for the transferred GUI and oracle events. We also discussed the factors impacting the effectiveness and efficiency of CraftDroid, which can be used as a guideline by researchers to improve test transfer techniques.

For the future work, we are planning to conduct empirical study with more apps and incorporate techniques such as crowd sourcing to improve the effectiveness of CraftDroid. We share the vision of Behrang and Orso [40] toward the establishment of a centralized repository similar to App Store, but for test cases. This Test Store will be able to generate feature-based test cases for newly developed apps. The knowledge mined from existing tests and apps, which we use for test transfer, can also have applications beyond testing, such as suggesting missing features and improving GUI layouts/flows for new apps.

Chapter 5

Inter-Platform Test Transfer

GUI testing is important for examining the end-to-end workflows and usability of GUI-based software. To reduce the manual effort of writing GUI tests, recent research has explored the potential of automatically reusing GUI tests by transferring them across similar applications. However, what is missing from the prior work is that such transfer may be required for apps available on different platforms. In particular, both web and Android are dominant platforms on which many organizations provide their software services. At the state-of-the-practice, even if the web and Android versions of an app provisioned by an organization substantially share the functionality, the developers have to manually write separate sets of tests for each version. This chapter proposes TransDroid, an automated tool that transfers GUI tests from a web app to its Android counterpart. Evaluation of TransDroid on real-world web and Android apps corroborates its effectiveness by achieving 77% success rate among the attempted transfers, along with 82% precision and 99% recall in the mapping of the GUI events and oracles.

5.1 Introduction

Usage-based GUI testing aims to cover the use-case scenarios of the software under test. Developers typically prefer usage-based GUI testing to other forms of GUI testing (e.g., crawling) that are use-case agnostic and simply aim for higher code coverage [90, 87]. Usage-based GUI testing provides the developers with actionable information that allows them to properly recreate the failures and debug their programs. However, this form of testing is tedious and time-consuming, since it often involves substantial manual effort of writing the test cases from scratch.

To reduce the manual effort of writing usage-based GUI tests, recent research has explored the possibility of reusing GUI tests by automatically transferring them across similar applications (apps) within a platform [114, 41, 73, 112, 42, 86]. By platform, we mean a particular computing domain, such as mobile, web, and desktop. A key insight guiding these efforts is that the GUI widgets of different apps providing the same functionality are semantically similar. As a result, it is possible to automatically generate a usage-based GUI test for a *target app* by reusing the test of a *source app*, provided that (1) both apps share the same feature (functionality); and (2) the correct mapping of the widgets between these two apps can be identified.

While the current techniques for intra-platform test transfer are promising, missing from the prior work is that such transfer may be required for apps on different platforms. In fact, many organizations provide their software services on multiple platforms. Case in point, among the top 50 most visited websites in the United States [3], 80% of them also provide native mobile apps for their users. Another example is WordPress [10], one of the most popular content management systems, which can be accessed via web browsers, mobile apps (Android and iOS), and desktop apps (Windows, MacOS, and Linux). At the state-of-the-practice, despite substantial overlap among several versions of an app provisioned by an organization and intended for execution on different platforms, developers have to manually write separate sets of tests for each version of app. We believe automated test transfer presents a promising solution in such settings, yet has never been explored in the past.

There are two main challenges in cross-platform test transfer that prior work has not addressed. The first is *incompatible actions*. Event synthesis is a necessary process for test transfer, in which appropriate actions such as a *click* are determined for the identified GUI widgets in the target app to compose executable events. This synthesis is guided by both the actions performed by the source test and the type of target widgets. While GUI-based apps share certain common actions such as *click* and *text input*, different platforms usually provide additional unique actions to optimize the user experience. As a result, if the source actions are platform-specific and not supported on the target platform, current techniques are not able to finish the transfer. For example, *mouseOver* is a common action in web testing for sub-menu exploration, but its corresponding action on Android is undefined.

The second challenge in cross-platform test transfer is *unclear widget context*. A core process in test transfer is to search and map the GUI widgets between the source and target apps. For example, what is most similar to the "Sign Up" button in the source app can be the "Register" button in the target app. In this case, a source GUI event clicking the "Sign Up" button can be transferred to a target event clicking the "Register" button. In prior work, the similarity between widgets are determined by their context, such as text values (e.g., "Register") and types (e.g., `Android.widget.Button`).

As part of the context, type information is important for the search and mapping of the widgets. For instance, if the source widget is an `Android.widget.Button`, the most similar target widget is likely also an `Android.widget.Button` (or at least a *clickable*). Nevertheless, such context may be missing or ambiguous when the transfer crosses platform boundaries. Take GUI widgets in web apps, i.e., HTML tags, as an example. There are two main categories of HTML tags: specific tags (e.g., `<a>`, `<textarea>`, and `<li>`) and generic tags (e.g., `<span>` and `<div>`). A characteristic of web apps is that, through registered JavaScript event handlers, the behavior of widgets can be easily changed or assigned. For example, developers can change the behavior of a `<textarea>` from *editable* to *clickable*. Furthermore, it is even more common to assign arbitrary behaviors to generic tags like `<span>`. In turn, context of source widgets on web may provide no or even wrong hints for the search and mapping of target widgets on, for instance, a mobile platform, like Android.

In this chapter, we propose TransDroid, an automated tool that addresses the aforementioned challenges in the context of web-to-Android test transfer. In other words, TransDroid transfers GUI tests from a web app to its Android counterpart. The reason for this implementation choice is the fact that the Internet era precedes the smartphone era [63, 72], and there are a large number of organizations developing their web app prior to their mobile app. Typical examples include WordPress [10], Wikipedia [11], Twitter [12], and Zoom [13]. As a result, we believe many organizations may benefit from TransDroid, allowing the tests created for their web app to be reused for their mobile app. Nevertheless, it should be noted that the aforementioned challenges are shared in all types of cross-platform test transfer, e.g., mobile-to-web, web-to-desktop, and we expect the overall approach described in this chapter to have application in other domains, albeit with a different implementation to account for platform-specific differences.

TransDroid has several key differences from prior work. First, it includes a pre-transfer phase with customizable action transformation rules to covert incompatible actions in the source test into compatible ones with the target platform. Moreover, besides widget context, TransDroid considers two other types of contextual information, i.e., *screen context* and *action context*. The inclusion of additional contextual information not only helps the search and mapping of the GUI widgets with unclear widget context, but also makes TransDroid capable of supporting the transfer of test steps or events that have no associated GUI widgets, e.g., *jumpByURL* event that navigates to a web page by directly changing the URL field in the browser.

We evaluated TransDroid with 20 real-world web and Android apps and 110 test cases, including 561 GUI and oracle events for the web apps. The experimental results show that 77% of the attempted transfers were successful, along with 82% precision and 99% recall for the widget mapping.

In short, this chapter makes the following contributions:

- A description of cross-platform test transfer problem and the associated challenges in the context of web-to-Android transfer.
- A novel approach to automatically transfer GUI tests from a web app to its Android counterpart. The tool implementing this approach is publicly available [14].
- An empirical evaluation on real-world apps demonstrating the effectiveness and efficiency of the proposed approach.

The rest of this chapter is organized as follows. Section 5.2 provides the background for understanding this work using a motivating example. Section 5.3 provides an overview of TransDroid as well as the implementation details of its components. Section 5.4 illustrates our proposed test generation algorithm. Section 5.5 presents the evaluation results. Section 5.6 discusses the limitations of this work. The chapter concludes with an overview of the related research and future work.

5.2 Background and Motivating Example

User interaction with GUI-based software is in terms of GUI events. A GUI event (w, a) consists of a widget w and an action a performed on w. Note that it is possible that there

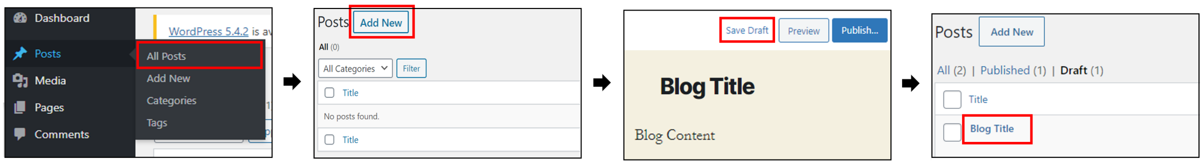


Figure 5.1: Saving a draft with the web app of WordPress

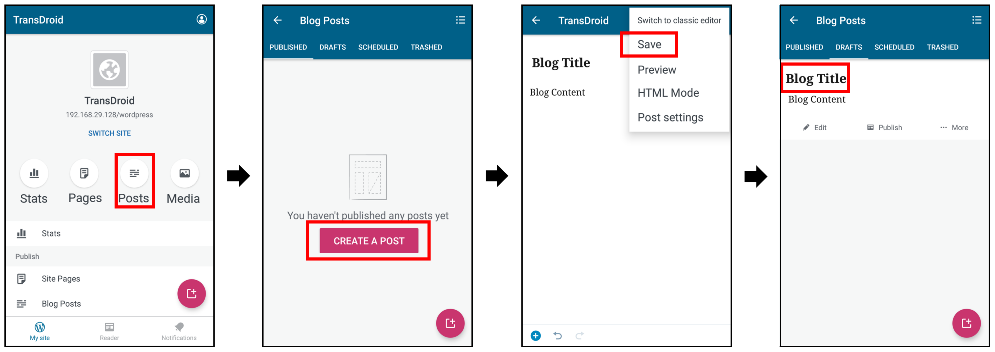


Figure 5.2: Saving a draft with the Android app of WordPress

is no widget associated with a GUI event, such as the *jumpByURL* event mentioned earlier. Moreover, an action in GUI events can be a simple operation (e.g., *button click*), or an operation with arguments (e.g., *text input*). Finally, if the action of a GUI event is an assertion, e.g., *isDisplayed*, we categorize the event as an *oracle event*.

To provide background knowledge about test transfer and illustrate the new challenges when the transfer is across platforms, consider WordPress [10], a popular blog and content management system. Figure 5.1 shows the excerpted steps to save a draft in WordPress using its web app. The user first gets to the post-listing page and then clicks the "Add New" button to initiate a new blog. After typing in the title and content, she clicks the "Save Draft" button and finally navigates back to the post-listing page to ensure the draft is saved. Figure 5.2 depicts how the same functionality is performed and tested on the Android app of WordPress. Table 5.1 shows the GUI and oracle events for this functionality on the two platforms.

As shown in Table 5.1, while the core steps to perform this functionality on these two

Table 5.1: The GUI and oracle events for saving a draft in WordPress on different platforms

Source Events on Web	Target Events on Android
1. ("Posts", *mouseOver*) 2. ("All Posts", *click*)	("Posts", *click*)
3. ("Add New", *click*)	("Create a Post", *click*)
4. ("Title", (*input*, "Blog Title"))	("Title", (*input*, "Blog Title")
5. ("Content", (*input*, "Blog Content"))	("Content", (*input*, "Blog Content"))
6. ("Save Draft", *click*)	("More options", *click*) ("Save", *click*)
7. ("", (*JumpByURL*, `all-posts.php`))	("Posts", *click*)
8. ("Draft", *click*)	("Drafts", *click*)
9. ("Blog Title", *isDisplayed*)	("Blog Title", *isDisplayed*)

platforms are conceptually identical, automatically reusing the source test for web app to generate the target test for Android app is hindered by several challenges. A critical challenge that has been addressed by prior work [86, 42] is the mapping of syntactically different but semantically similar GUI widgets between apps, such as the "Add New" and "Create a Post" buttons in Table 5.1. By leveraging advances in natural language processing (described in Section 5.4), prior work has shown the possibility of resolving such non-trivial mappings to transfer the GUI events.

Nevertheless, prior techniques have not addressed several challenges that are unique to cross-platform transfer. First, here the source test contains actions that do not exist on the Android platform, i.e., *mouseOver* and *jumpByURL*. Second, sometimes the contextual information about the source widgets are insufficient for guiding the search and mapping of the target widgets. For instance, if the source widget is a `<span>` HTML tag with text "Posts", and there are two target widgets, an `Android.widget.TextView` with text "Blog Posts" and an `Android.widget.Button` with text "Create a Post" (just as shown in the second screen of Figure 5.2), it is difficult to determine which one is the corresponding target widget. In other words, the behavior of the source tag (i.e., `<span>`) is unclear and other attributes, such as text values, are insufficient for determining the proper target widget.

We have designed TransDroid to overcome the aforementioned challenges and transfer such tests from web to Android. For example, the first *mouseOver* event and the second

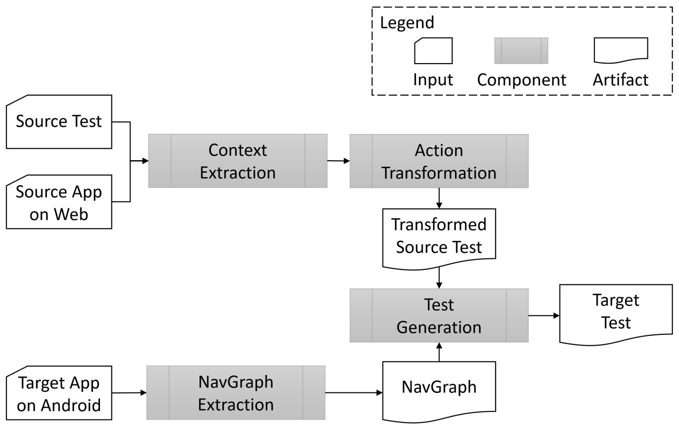


Figure 5.3: Overview of TransDroid

click event in the source test are merged, and then transferred as the first *click* event on the target app. Furthermore, the *jumpByURL* event in the source test is first converted to an intermediate event *navigateToActivity* on Android, and then transferred as the *click* event on the target app.

5.3 Approach

Figure 5.3 provides an overview of TransDroid. It takes a source test, a source app, and a target app as input, and generates a target test that examines the same functionality as the source test on the target app. TransDroid consists of four components: *Context Extraction*, *Action Transformation*, *NavGraph Extration*, and *Test Generation*. We describe the implementation of each component in the following subsections.

5.3.1 Context Extraction

Context Extraction component execute the source test and retrieves the contextual information related to each event in the source test. Like prior work [86, 42], we extract *widget* *context* that comes from the attributes of the GUI widgets interacted by the source test. However, unlike prior work, we additionally extract two other types of contextual information: *screen context* and *action context*. Screen context comes from the attributes of the GUI screens visited by the source test. Action context simply comes from the actions in the source test.

While we share the same insight as prior work that widget context can help identify correct target widget in test transfer, we believe that screen and action contexts, which are missing from the prior work, are also important to address the new challenges posed by cross-platform transfer. If we perceive the execution of a test as traversal through a graph consisting of an app's GUI screens, screen and action contexts provide additional information about how the path is visited. Following this insight, including the screen context in our analysis allows us to support source events that have no associated GUI widgets, such as the *jumpByURL* event mentioned previously. Furthermore, including action context in our analysis allows us to supplement our knowledge of the behaviorally ambiguous widgets. Taking the `<span>` tag described in Section 5.2 as an example, if its accompanied action is *click*, the search for the target widgets on Android can be limited to *clickables*. On the other hand, if the accompanied action is *input*, the search for the target widgets can be limited to *editables* such as `Android.widget.EditText`. Our transfer algorithm, thus, relies on all three forms of contextual information for search and mapping of the widgets.

In TransDroid, the Context Extraction component is implemented for web apps. It instruments and executes the source test to retrieve the contextual information related to each event. The widget context for GUI widgets in web apps, i.e., HTML elements, comes from their attributes, such as *id*, *name*, *class*, *href*, *placeholder*, etc., as well as the enclosed text. Moreover, the screen context in web apps are the title of the page (i.e., `<title>` tag) and first header element (e.g., `<h1>` tag), since they indicate the primary semantics of the screens. The action context in web apps are the actions performed by the source test, such as *click*,

```
{
  "widget-context": {
    "class": "wp-menu-name",
    "text": "Posts"
  },
  "screen-context:": {
    "title": "Dashboard",
    "h1": "Dashboard"
  },
  "action-context": "mouseOver"
}
```

Figure 5.4: Retrieved contexts of (“Posts”, *mouseOver*) in Table 5.1

input and *mouseOver*. For example, Figure 5.4 shows the retrieved contexts of the first event in Table 5.1, i.e., (“Posts”, *mouseOver*).

5.3.2 Action Transformation

Action Transformation component processes the source test and ensures that the actions in the transformed source test are compatible with the target platform.

The transformation relies on a set of customizable rules that perform one of the four basic operations: *reuse*, *merge*, *conversion*, and *removal*. First, the actions commonly shared by GUI-based software such as *click* can be directly reused on the target platform. Next, if an event contains a platform-specific action, it is possible to combine the event with its preceding or succeeding event (i.e., merge). Alternatively, we may replace the action with a similar action available on the target platform (i.e., conversion). Finally, if the incompatible action is not suitable for merge or conversion, the event may be removed from the transformed test.

Table 5.2 shows the action transformation rules adopted by TransDroid. An example of merge action is *mouseOver*, since this action is typically followed by *click* to form a common combo operation on web apps to open a sub-menu. Therefore, a *mouseOver* event, together with the retrieved context, is merged with the following *click* event.

Table 5.2: Action Transformation Rules in TransDroid

Source Action on Web	Target Action on Android	Operation Type
click()	*click()*	reuse
textInput()	*textInput()*	reuse
mouseOver()	(merge into the next source action)	merge
rightClick()	(merge into the next source action)	merge
jumpByURL()	*navigateToActivity()*	conversion
doubleClick()	*click()*	conversion
keyDown()	(removed)	removal
switchToWindow()	(removed)	removal

Examples of the converted actions include *jumpByURL* and *doubleClick*. Because the semantics behind *jumpByURL* is a change of GUI state, this action is converted to *navigateToActivity*, an intermediate action defined in TransDroid for Android with the similar intention. On the other hand, *doubleClick* in web apps is usually adopted to provide desktop-like user experience for features, such as opening a file in file manager or editing cells in a spreadsheet. Since such a user experience is rarely available in native mobile apps, we simply change *doubleClick* to *click*.

Finally, an instance of removed actions is *keyDown*, since it is usually used to perform a modifier key press (e.g., Shift) and may be safely removed without affecting the testing flow of the generated target test. Note that the presented action transformation rules can be extended for more platform-specific actions, or customized to accommodate the context of the target apps.

5.3.3 NavGraph Extraction

NavGraph Extraction component extracts the *Navigation Graph* of the target app. A Navigation Graph G of an app A is generally defined as a tuple (s, V, E) where:

- s denotes the starting state, i.e., the initial state after A has been fully loaded and started.

- V is a set of GUI states (screens). Each $v \in V$ represents a unique runtime GUI state in A. Moreover, each v is associated with a set of GUI widgets, W_v, that could be rendered in state v.
- E is a set of edges between the GUI states. Each $e = (v_1, v_2, (w, a)) \in E$ represents a transition from v_1 to v_2 by firing a GUI event (w, a).

We leveraged and modified the static analysis tool in prior work [86] to construct the Navigation Graph for the target app. The tool first extracts Activities in the target app as the GUI states. For each Activity, it then retrieves the associated GUI widgets (as well as their context if possible) from the resource files and source code. At last, it identifies transitions among Activities as the edges, by analyzing the registered event handlers on the widgets associated with each Activity. Figure 5.5 illustrates part of the Navigation Graph for WordPress on Android. This graph provides information about the screens and widgets comprising the target app to the Test Generation component, which as described next, applies a novel, heuristic-based algorithm to generate the target tests.

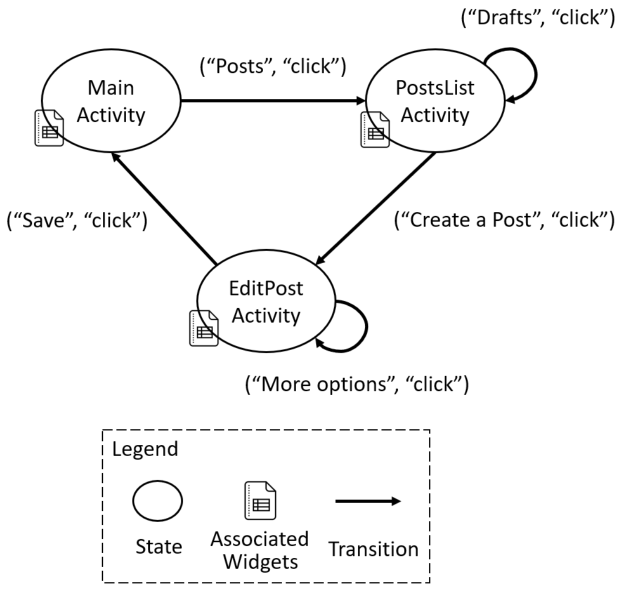


Figure 5.5: Excerpted Navigation Graph for WordPress on Android

5.4 Test Generation

Test Generation component takes a *targetApp* and its corresponding *navGraph*, as well as a transformed source test t as input, and generates the target test t_n using a model-based, greedy search algorithm, as described in Algorithm 3. The algorithm consists of three main steps: (1) transfer the source events one-by-one to the target app; (2) update the Navigation Graph based on runtime information; and (3) repeat the transfer until no improvement can be made.

First, for each $event = (w_i, a_i) \in t$, if the event has an associated widget, i.e., w_i is not *null*, Algorithm 3 consults *navGraph* with the widget context and action context of the event, and collects a list of widgets in the target app, *widgets*, in which the widgets are ranked based on their similarity to w_i (line 6-9). Next, for each $w_n \in widgets$, it checks whether w_n is reachable, and if so, identifies a sequence of events, *leadingEvents*, that should be executed to reach w_n (line 10-12). After that, it determines an appropriate action a_n for the identified w_n, and composes the $targetEvent = (w_n, a_n)$ (line 13-14), which will be added into t_n together with *leadingEvents* (line 28). More details about similarity computation (lines 9 and 20), reachability check (lines 11 and 22), and action generation (line 13) are described in the next subsections.

On the other hand, if the source event has no widget attached, i.e., w_i is *null*, that means no target widget needs to be mapped. Instead, we consult *navGraph* with the screen context of the event, and try to identify a reachable GUI screen s in the target app that is most similar to the screen visited by the source event (lines 18-26). In this case, *leadingEvents* is a sequence of events that should be executed to reach s, (line 23) and *targetEvent* is left to be *null*.

Once all the events in t are processed, the algorithm computes the fitness of the generated test t_n by evaluating its similarity to source test t, i.e., a weighted average of similarity scores (described in the next subsection) is computed for the corresponding events. If the fitness of t_n cannot be improved any further by a user-specified threshold or other termination criterion such as time limit is reached (line 30), the algorithm terminates and returns (line 34). Otherwise, it repeats the transfer and tries to find a better solution with an updated *navGraph*. As described in the following subsections, during the reachability analysis (lines 11 and 22), we also update *navGraph* to improve the precision of our statically-retrieved app model with dynamically observed behaviors, thereby improving the likelihood of solving the search problem with each iteration of the algorithm.

5.4.1 Similarity Computation

Similarity between GUI widgets or screens is primarily determined by their context. The similarity between two widgets is based on their widget context and action context. For two GUI screens, the similarity is determined by their screen context. Since the contexts are represented as words or word lists, the similarity can be computed by leveraging different metrics in natural language processing (NLP). In this work, following the recent test transfer works [42, 86], we leverage Word2Vec [101] to compute the similarity between two widgets or GUI screens. Word2Vec is a neural network model that captures the linguistic contexts of words. In this model, each word is represented as a real-value vector (called word embedding). A characteristic of Word2Vec is that semantically related words are close together in terms of their cosine similarity. For instance, "Create" is closer to "Add" (with cosine similarity of 0.47) than "Delete" (with cosine similarity of 0.33) in the vector space. As a result, even if the "Add New" button does not exist in the Android App, TransDroid can still find its most similar widget, i.e., the "Create a Post" button, as shown in the motivating example (the second row in Table 5.1).

To exemplify how we compute the similarity between two contexts, consider the "Add New"

Algorithm 3 TransDroid: Test Generation

```
Input:
    $targetApp$, $navGraph$, and
    Transformed source test $t = \{(w_1, a_1), (w_2, a_2), ...\}$
Output:
    $t_n = \{(w_{n_1}, a_{n_1}), (w_{n_2}, a_{n_2}), ...\}$ for $targetApp$

 1: while $true$ do
 2:     $t_n = \emptyset$
 3:     for each $event = (w_i, a_i) \in t$ do
 4:         $leadingEvents = \emptyset$
 5:         $targetEvent = null$
 6:         if $w_i$ is not $null$ then
 7:             $x_w = getWidgetContext(event)$
 8:             $x_a = getActionContext(event)$
 9:             $widgets =$
                    $getSimWidgets(x_w, x_a, navGraph)$
10:             for each $w_n \in widgets$ do
11:                 if $isReachable(w_n, t_n, navGraph)$ then
12:                     $leadingEvents =$
                            $getPath(w_n, navGraph)$
13:                     $a_n = generateAction(w_i, a_i, w_n)$
14:                     $targetEvent = (w_n, a_n)$
15:                     break
16:                 end if
17:             end for
18:         else
19:             $x_s = getScreenContext(event)$
20:             $screens = getSimScreens(x_s, navGraph)$
21:             for each $s \in screens$ do
22:                 if $isReachable(s, t_n, navGraph)$ then
23:                     $leadingEvents =$
                            $getPath(s, navGraph)$
24:                     break
25:                 end if
26:             end for
27:         end if
28:         $t_n = t_n \cup leadingEvents \cup targetEvent$
29:     end for
30:     if $\Delta\ fitness(t, t_n) \leq threshold\ or\ timeout$
31:         break
32:     end if
33: end while
34: return $t_n$
```

button and the “Create a Post” button in the motivating example (the second row in Table 5.1). To compute the similarity between their text, i.e., “Add New” and “Create a Post”, we first apply a series of common practices in NLP, including tokenization and stopword removal, to convert the text into word lists, i.e., [“Add”, “New”] and [“Create”, “Post”]. Next, we query a pre-trained Word2Vec model released by Google [15] to obtain the cosine similarity scores for the word pairs as follows:

$$\begin{array}{c|cc|} & \mathit{Add} & \mathit{New} \\ \hline \mathit{Create} & \mathbf{0.47} & 0.22 \\ \mathit{Post} & 0.10 & \mathbf{0.12} \end{array}$$

The similarity for the text is then calculated as $(0.47 + 0.12)/2 = 0.29$, the average of the pairs with the highest score. We compute the similarity scores for other attributes of these two buttons following the same way. The final similarity score is calculated as a weighted sum of the scores from all of their attributes.

5.4.2 Reachability Check

The Navigation Graph needs to be verified and updated during transfer, because the graph may be initially derived through static analysis, which tends to over-approximate the app’s runtime behavior. Algorithm 4 describes the function `isReachable` called on lines 11 and 22 of Algorithm 3. This function serves two main purposes: (1) check if an *entity*, i.e., a widget or GUI screen, is reachable by verifying the possible paths leading to it; and (2) update *navGraph*, including the events that trigger transitions between screens as well as the associated widgets with each GUI screen, during the verification.

To that end, the function first restarts the app and executes t_n, i.e., the events successfully transferred so far, to get to the current GUI screen, *curScreen* (line 2-3). Next, if *entity* is a

Algorithm 4 TransDroid: Function: isReachable()

```
1: function isReachable($entity, t_n, navGraph$)
2:     $execute(t_n)$
3:     $curScreen = getCurrentGUIScreen()$
4:     if $entity$ is a widget then
5:         $dstScreen = getGUIScreen(entity, navGraph)$
6:     else                                          ▷ $entity$ is a GUI Screen
7:         $dstScreen = entity$
8:     end if
9:     $paths = getPaths(curScreen, dstScreen, navGraph)$
10:    for each $path \in paths$ do
11:        $isValid = validate(entity, path, navGraph)$
12:        if $isValid$ is $true$ then
13:            return $true$
14:        end if
15:    end for
16:    return $false$
17: end function
```

widget, the function assigns the GUI screen associated with *entity* to the destination screen, *dstScreen* (line 5); otherwise the *entity* itself is assigned to *dstScreen* (line 7). After that, all possible paths between *curScreen* and *dstScreen* are executed to verify whether *entity* is reachable. The function returns *true* once a feasible path for *entity* is found; otherwise it returns $false$ (line 9-16). Moreover, the function `validate` in line 11 updates *navGraph* by (1) removing unreachable paths; and (2) adding newly encountered widgets at runtime to the associated widgets of the GUI screen.

5.4.3 Action Generation

The function `generateAction` in line 13 of Algorithm 3 determines a proper action a_n for the identified target widget w_n. Typically, if the source event (w_i, a_i) contains a generic action such as *click* or *text input*, the action can be directly reused, i.e., $a_n = a_i$. However, it is possible that the correct action for a_n is other type of actions supported by or registered on w_n, such as *longClick*. On the other hand, a_n can be an assertion (e.g., *isDisplayed*) if the source event is an oracle event. Therefore, the implementation of `generateAction` needs to

Table 5.3: Assertion types supported by TransDroid. (w_i, a_i): source oracle event. (w_n, a_n): transferred target event.

a_i	a_n
$isAttrEqual(VALUE_i, attr(w_i))$	$isAttrEqual(VALUE_n, attr(w_n))$
$isDisplayed(w_i)$	$isDisplayed(w_n)$
$textPresent(STRING)$	$textPresent(STRING)$
$textNotPresent(STRING)$	$textNotPresent(STRING)$

accommodate these situations.

In our implementation, for GUI events, we first analyze the source code[1] to check whether the identified target widget has specific event listeners, such as *setOnLongClickListener()*, and if so, we assign the action corresponding to such an event listener as the target action a_n. Otherwise, we reuse the source action.

If the source event (w_i, a_i) is an oracle event, i.e., a_i is an assertion, this function needs to be customized for different types of assertion. Currently, TransDroid supports four types of assertion commonly used in web testing [16], as shown in Table 5.3. The first two assertion types, *isAttrEqual* and *isDisplayed*, are widget-specific, and their arguments needs to be modified when transferred to the target app. The other two assertion types, *textPresent* and *textNotPresent*, are widget-irrelevant assertions and can be directly transferred to the target app.

5.4.4 Walk-Through of the Motivating Example

We provide a walk-through of how TransDroid transfers the test shown in our illustrative example (recall Figures 5.1 and 5.2) to help the reader see the entire framework in action. First, Context Extraction executes the source test shown in Table 5.1 on the source app to retrieve the contextual information related to each event, and annotates the source test with this information. After that, Action Transformation parses the source test and transforms

[1]Our analysis is actually performed on decompiled binary code (i.e., APKs) using Soot, a static analysis framework for Java [127].

it to an Android-compatible test, i.e., a test in which all of the actions are supported in Android. Particularly, the first *mouseOver* event is merged into the following *click* event, and the *jumpByURL* event is converted to *navigateToActivity* event. At the same time, NavGraph Extraction statically retrieves the Navigation Graph of the target app as the input for Test Generation.

In Test Generation, each event in the transformed source test is transferred one-by-one. The target widgets or GUI screens most similar to the source contexts are identified from the Navigation Graph with our formula for computing similarity. For example, when transferring the *navigateToActivity* event (transformed from *jumpByURL*), the algorithm searches for an Android Activity that is most similar to the screen context retrieved from `all-posts.php` (i.e., Activity with a name that is most similar to title/heading of php file), and generates the events leading to that screen. This results in a click on "Posts" button, which initiates a transition from *MainActivity* to *PostsListActivity*, as shown in Figure 5.5. All other events are similarly transferred. Finally, the last oracle event in the source test, i.e., existence check for the `<a>` tag with text *"Blog Title"*, is transferred to an existence check for the `android.widget.TextView` with the same text.

5.5 Evaluation

We investigate the following research questions in our experimental evaluation of TransDroid:

RQ1. How effective is TransDroid in terms of (1) the precision and recall for widget mapping, and (2) the number of successful transfers compared to total attempted transfers?

RQ2. What are the main reasons behind transfer failure?

Table 5.4: Subject apps and test suites

Subject App	Description	Web Version	Android Version	#Web Tests	#Events in Web Test		
					GUI	Oracle	Total
BuzzFeed	News and entertainment	Live website	com.buzzfeed.android:v2021.3	11	30	19	49
DokuWiki	Collaborative editor	v2018-04-22	com.fabienli.dokuwiki:v0.10	9	29	17	46
Etsy	E-commerce	Live website	com.etsy.android:v5.53.1	13	40	18	58
Fox News	News television channel	Live website	com.foxnews.android:v4.22.0	11	30	17	47
GitLab	DevOps lifecycle tool	v13.2.2	com.commit451.gitlab:v2.6.3	11	38	27	65
Groupon	E-commerce	Live website	com.groupon:v20.10.224420	13	39	23	62
Hacker News	Social news forum	Live website	net.dreambits.hackernews:v2.5	12	39	24	63
OwnCloud	File hosting	v10.5	com.owncloud.android:v2.15	11	40	20	60
Wikipedia	Online encyclopedia	Live website	org.wikipedia:v2.7.50320	9	37	14	51
WordPress	Content management	v5.3.2	org.wordpress.android:v14.3	10	43	17	60
Total				110	365	196	561

RQ3. How much effort can be saved by using TransDroid to generate tests?

RQ4. How efficient is TransDroid in terms of the running time to perform cross-platform transfer?

5.5.1 Experimental Setup

We implemented TransDroid with Python and Java for web tests written using Selenium [17]. Existing test cases for the subject apps are written with Selenium's Python client. The transferred Android tests are stored in JSON format and executed by our test runner implemented with Appium [8]. In our experiments, we used ChromeDriver [18] to execute the web tests, and a Pixel 2 Emulator running Android 7.1 (API 25) for test generation. The experiments were conducted on a Windows laptop with 2.8 GHz Intel Core i7 CPU and 32 GB RAM. Our experimental data is publicly available [14].

Subject apps and test suites. As noted by others [102, 33], it is very difficult to find publicly available web apps that have working UI test suites. We managed to find 10 pairs of web and Android apps (20 in total) with either existing tests, or existing documentation containing the test descriptions. We first collected web tests for each subject app, if there were any, and then augmented the test suites with the test cases corresponding to the found test descriptions. Table 6.1 shows the 10 pairs of real-world subjects used in our study, including the size of test suites and the number of GUI and oracle events. There are 110 web tests in total, each containing 5.1 events (including 1.8 oracle events) on average. In addition, we manually constructed the corresponding Android tests. These tests served as the ground truth in our experiments to determine if (1) the target widgets were correctly identified, and (2) the transfers were successful or not (detailed in the next paragraph).

Effectiveness of attempted transfers. TransDroid transfers each of the tests for a web app to its Android counterpart, resulting in 110 total attempted transfers. For each transfer, we used the ground truth to examine the generated test to identify *false positive*, *false negative*, and *true positive* cases as follows: *false positive* occurs when the target widget of manual transfer is different from the widget identified by TransDroid; *false negative* occurs when TransDroid fails to find a target widget, while manual transfer can; and *true positive* occurs when the target widget from manual transfer matches the widget identified by TransDroid. Based on these metrics, we measured the *Precision* as the number of generated target events that are correct. Additionally, *Recall* measures how many of the source events are correctly transferred.

Precision and recall can faithfully evaluate the correctness of widget mapping, but not necessarily the successfulness of test transfer [86, 133]. In other words, precision and recall do not consider whether the generated tests are executable or applicable in the context of the target app. For example, suppose a web app requires the user to provide a password only once during registration, but twice on its Android counterpart for confirmation. In that case, the transfer may have very high precision and recall if most of the source events are correctly migrated. However, the generated test on Android is still not executable because it lacks the required password confirmation step. As a result, we also report whether the attempted transfers were successful by manually examining the generated tests. A successful transfer means that the generated test was executable and actually meaningful in the context of the target app, verifying the same feature as the source test.

Table 5.5: Effectiveness evaluation of TransDroid

Subject	GUI Event		Oracle Event		#Successful
	Precision	Recall	Precision	Recall	Transfer
BuzzFeed	64%	95%	63%	100%	64% (7/11)
DokuWiki	70%	90%	94%	94%	89% (8/9)
Etsy	95%	100%	94%	100%	100% (13/13)
Fox News	86%	100%	71%	100%	64% (7/11)
GitLab	81%	100%	81%	100%	64% (7/11)
Groupon	74%	100%	87%	100%	69% (9/13)
Hacker News	79%	100%	83%	100%	67% (8/12)
OwnCloud	71%	96%	85%	100%	73% (8/11)
Wikipedia	86%	100%	92%	92%	89% (8/9)
WordPress	88%	100%	100%	100%	100% (10/10)
Total	80%	99%	85%	99%	77% (85/110)

Effort Reduction. Another perspective to evaluate the usefulness of TransDroid in practice is to measure how much effort developers can save if they adopt this tool to generate tests instead of writing them from scratch, regardless of whether the transfers are successful or not. To that end, we first measure how close a transferred test is to its ground-truth test by computing their Levenshtein distance [85] or edit distance. Levenshtein distance is a string metric to compute the minimum number of edits required to change one word to another word. In our case, a single edit is defined as an insertion, deletion or substitution of an event in the transferred test. Next, we further define *reduction of effort* as follows:

$$Reduction(t_n) = 1 - \frac{editDistance(t_n, t_g)}{\#events(t_g)}$$

This equation measures the manual effort reduced by a generated test t_n compared to writing its ground truth t_g from scratch. For example, if a 6-event generated test needs 2 edits (e.g., 1 deletion and 1 substitution) to its 5-event ground truth, compared to writing the ground truth from scratch, the reduced manual effort through the generated test is $1-(2/5) = 60\%$.

5.5.2 RQ1: Effectiveness

Table 5.5 demonstrates the effectiveness of TransDroid in terms of precision, recall, and successful transfers for each subject listed in Table 6.1. These results show that in total, 77%

of the attempted transfers by TransDroid are successful, with an overall 82% precision and 99% recall considering all the transferred GUI and oracle events. TransDroid is substantially effective in identifying correct GUI widgets and successfully transferring tests from web to Android.

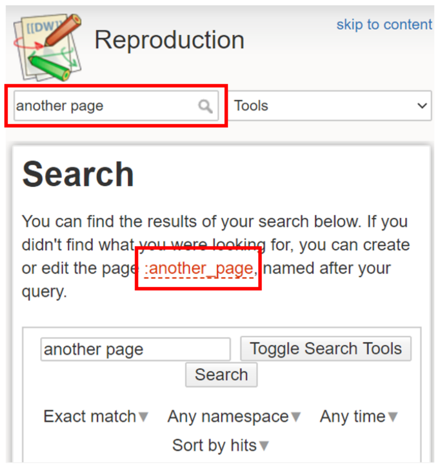
Interestingly, we found that perfectly matching all the widgets and screens in a source test is not always necessary to successfully transfer the test. Two instances for such cases are WordPress and Etsy, in which all the tests were successfully transferred (i.e., 100% success rate), despite the existence of several false positives in the matched GUI events (i.e., imperfect precision). The reason is that sometimes false positives are not harmful, since the same feature may be implemented differently on two platforms. For example, to access the "About Me" page on WordPress's web app, users need to expand and navigate the side menu, which is not required on the Android app, as it provides a shortcut to that page on its main screen.

Another important observation from the results in Table 5.5 is that the success rate varies among different subjects, ranging from 64% (on BuzzFeed, Fox News, and GitLab) to 100% (on Etsy and WordPress). In the next research question, we investigate the attributes that impact the success rate of test transfer.

5.5.3 RQ2: Factors Impacting Effectiveness

We manually investigated all of the attempted transfers, including both successful and failed ones, to identify the factors that impact the effectiveness.

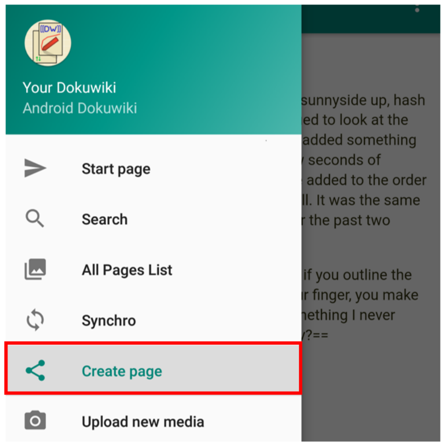
Insufficient widget context. Insufficient contextual information in the target widgets, such as indistinguishable or missing textual information, impacts TransDroid's ability to find a proper match. For instance, while the input fields for shipping address in Groupon's web app contain distinguishable identifiers such as `city` and `zip-code`, the Android app



Search

You can find the results of your search below. If you didn't find what you were looking for, you can create or edit the page :another_page named after your query.

(a) Create a page on the web app through the search bar and a dynamically generated link



(b) Create a page on the Android app through a button

Figure 5.6: Different interaction flows to create a page on Dokuwiki across platforms

uses the same identifier, i.e., `edit-text`, for all of the corresponding fields. As a result, TransDroid failed to transfer the tests dealing with the shipping feature. Another example is the Navigation Drawer (a.k.a., the menu icon or hamburger icon) in GitLab's Android app. It is implemented as an image button, rather than a native Android icon, without any associated textual information. As TransDroid only considers textual information for widget context, this button could not be matched and the features accessible through this button remain undiscovered.

Missing features. The test transfer fails if the tested feature is not implemented in the target app/platform. For example, the web users of OwnCloud can restore deleted files from the recycle bin, but this feature is not provided on OwnCloud's Android app. Another instance of such transfer failure is the "unvote" feature (i.e., to revoke the vote for a news post) that is only provided on the web app of Hacker News, and not its Android counterpart.

Radically different interaction flows. If the interaction flow of accessing a feature is utterly different across platforms, the transfer may fail. For example, to create a new page on DokuWiki's web app, users need to first search for the name of the page that they want

Table 5.6: Average reduction of effort

Subject	#Events on Average			%Reduction
	Generated Test	Ground Truth Test	Edit Distance	
BuzzFeed	5.13	4.13	1.75	58%
DokuWiki	6.11	5.89	0.78	87%
Etsy	4.62	4.46	0.23	95%
Fox News	4.89	4.67	0.67	86%
GitLab	6.27	6.09	1.27	79%
Groupon	5.23	4.85	1.08	78%
Hacker News	6.82	6.55	1.00	85%
OwnCloud	6.80	6.30	1.50	76%
Wikipedia	6.33	5.89	0.89	85%
WordPress	6.50	5.60	0.90	84%
Total	5.84	5.44	0.98	82%

to create, and then click the link dynamically generated in the search result. This form of interaction, however, is not supported by the Android app. A new page can only be created by clicking the "Create page" button on the Android app, as shown in Figure 5.6.

Test length is NOT a key factor. Prior work in intra-platform test transfer [86] found a strong negative correlation between test length (i.e., number of total events) and the effectiveness metrics (i.e., precision, recall, and success rate). Their finding inspired us to investigate if a similar correlation can be found in inter-platform test transfer. We conducted a Pearson correlation analysis [9] on our dataset, and observed a negligible correlation with the coefficients ranging between 0.03 and 0.30. Since it appears test length is not a key factor impacting effectiveness of cross-platform test transfer, we argue that future research should focus on other factors to improve the success rate of cross-platform transfer.

5.5.4 RQ3: Reduced Effort

Table 5.6 demonstrates the average number of events comprising the ground-truth tests and transferred tests, along with their edit distance. The results show that TransDroid can save 82% of the manual effort on average, compared to writing the ground-truth tests from scratch. Taking WordPress as an example, on average the tests generated by TransDroid

Table 5.7: Efficiency evaluation of TransDroid

Subject	Time in Sec for a Transfer		
	Min	Max	Avg
BuzzFeed	48	3160	518
DokuWiki	14	2060	396
Etsy	17	271	95
Fox News	50	1724	639
GitLab	13	157	66
Groupon	54	644	199
Hacker News	21	424	133
OwnCloud	42	3132	629
Wikipedia	51	4445	769
WordPress	18	613	212
Total	13	4445	349

contain 6.5 events and need only 0.9 manual edits to be transformed to the ground truth, thereby achieving a substantial reduction in the manual effort of creating the tests from scratch. This result hints at the potential utility of TransDroid, even when it fails to successfully transfer the entire test suite.

5.5.5 RQ4: Efficiency

Table 5.7 shows the execution time for the attempted transfers in our experiments. On average, a test transfer takes less than 6 minutes, ranging from 13 seconds to 1.2 hours among the different tests and subject apps. Note that, however, among all 110 attempted transfers, only 4 of them took more than half an hour (the four largest numbers shown in Table 5.7). The average execution time for the other 106 transfers is only 241 seconds or approximately 4 minutes. While it is not feasible to directly compare our inter-platform transfer work against prior works targeting intra-platform transfer, the efficiency demonstrated by TransDroid is quite impressive, since prior work for intra-platform transfer takes 1.5 hours on average [86] to finish a transfer.

We investigated the four longest transfers and found that they spent most time in checking the reachability of many candidate widgets. That is, they ran the function `isReachable` in

Algorithm 4 repeatedly. This is the most time-consuming element of the Test Generation component in general, as it frequently restarts the target app to validate the potential paths for a candidate widget or screen. Nevertheless, such reachability checks may be accelerated if executed in parallel with multiple devices or emulators.

5.6 Assumptions and Limitations

Similar to all prior work in intra-platform test transfer [114, 41, 73, 112, 42, 86], TransDroid assumes the source web app and its Android counterpart have similar features. If not, test transfer would not work. In some cases the Android app may have unique and platform-specific features, such as notification- or geolocation-related functionalities, for which test transfer will not be possible, since the corresponding web app lacks those features and hence does not have any related tests for transfer. The goal of this work is to reduce the manual effort of writing tests for features that are shared.

TransDroid also assumes the interaction flows between the web and the Android versions of an app are similar, albeit not identical, for a given feature. As discussed in RQ2 (Section 5.5.3), we acknowledge that the same feature may be realized with drastically different interaction flows on different platforms. Nevertheless, as our evaluation shows, that is typically not the case in practice, and the proposed transfers are effective on a considerable number of apps that have similar cross-platform behaviors.

The current implementation of TransDroid does not support some actions in web testing such as drag and drop. Moreover, this work does not consider external communications in the source web test, such as choosing a local file or login with OAuth. Such limitations could be addressed by extending the current action transformation rules. That said, one can trivially construct an automated pre-processing phase to exclude the source tests containing unsupported actions to improve the success rate of transfers.

5.7 Conclusion

Automated test transfer is a promising method of generating high-quality tests for verification of similar features among mobile apps. In this chapter, we described the cross-platform test transfer problem and the associated challenges that prior works have not addressed. We presented TransDroid, an automated tool for solving this problem in the context of web-to-Android transfer. TransDroid adopts novel transformation techniques and test generation algorithms to resolve the challenges of overcoming the incompatibilities between platforms. Our evaluation on real-world apps demonstrated the effectiveness and efficiency of TransDroid, as it successfully transferred 77% of the test cases in our experiments. Our results indicate that even when test transfer is not completely successful, it has the potential of significantly reducing the manual effort of creating tests for similar apps, i.e., 82% reduction on average in our study subjects.

We also discussed the factors impacting the effectiveness of TransDroid, addressing which will frame part of our future work. We also aim to conduct a user study involving real developers to further validate our empirical findings. Finally, we plan to investigate the application of TransDroid for test transfer among other platforms, such as mobile-to-web and web-to-desktop.

Chapter 6

Feature-Based Test Augmentation

Core features (functionalities) of an app can often be accessed and invoked in several ways, i.e., through alternative sequences of user-interface (UI) interactions. Given the manual effort of writing tests, developers often only consider the typical way of invoking features when creating the tests (i.e., the "sunny day scenario"). However, the alternative ways of invoking a feature are as likely to be faulty. These faults would go undetected without proper tests. To reduce the manual effort of creating UI tests and help developers more thoroughly examine the features of apps, we present ROUTE, an automated tool for feature-based UI test augmentation for Android apps. ROUTE first takes a UI test and the AUT as input. It then applies novel heuristics to find additional high-quality UI tests, consisting of both inputs and assertions, that verify the same feature as the original test in alternative ways. Application of ROUTE on several dozen tests for popular apps on Google Play shows that for 89% of the existing tests, ROUTE was able to generate at least one alternative test. Moreover, the fault detection effectiveness of augmented test suites in our experiments showed substantial improvements of up to 39% over the original test suites.

6.1 Introduction

Feature-based user-interface (UI) testing serves as a primary way of examining the end-to-end workflows and usability of interactive applications such as mobile apps. Compared to other forms of UI testing (e.g., crawling or random exploration) that simply focus on achieving higher code coverage, feature-based UI testing aims to cover the features (functionalities) of the app under test (AUT). A prior study has showed that this type of testing is preferred by mobile app developers [90]. While it is straightforward to conduct usage-based testing manually, developers often choose to write scripted or automated UI test cases to make such testing repeatable in the context of continuous integration [58].

Automated feature-based testing has several advantages: reliability, execution speed, and in particular, the manifestation of developers' knowledge regarding software functionality through oracles, i.e., assertions in test cases. However, writing the tests involves substantial manual effort; thus, developers in practice only create a limited number of such tests due to time constraints. A recent study about test automation in open-source Android apps shows that within the real-world projects adopting automated UI testing, half of them contain fewer than 8 UI test cases [87].

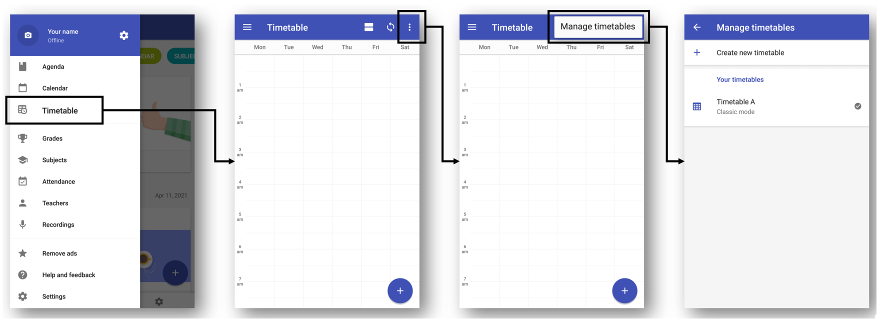
Furthermore, the functionalities examined by automated feature-based tests are usually important, core capabilities of an app that can be accessed in multiple ways, i.e., through alternative sequences of UI interactions. The existing automated test generation techniques [1, 35, 37, 48, 70, 94, 95, 97, 103, 128, 123, 57] cannot generate high-quality tests, consisting of both inputs and proper assertions, as they lack developers' knowledge; instead, they generate inputs for exploring apps without providing proper assertions to verify the resulting behavior. On the other hand, when creating the automated test scripts, developers may only consider the typical way of invoking a feature (i.e., the "sunny day scenario" or "happy path"), neglecting alternative ways of invoking it.

```
find_element_by_id("school/timetable").click()
find_element_by_id("more_options").click()
find_element_by_id("school/item_1").click()
e = find_element_by_id("school/title")
assertEquals(e.getText(), "Manage timetables"); // Oracle
```

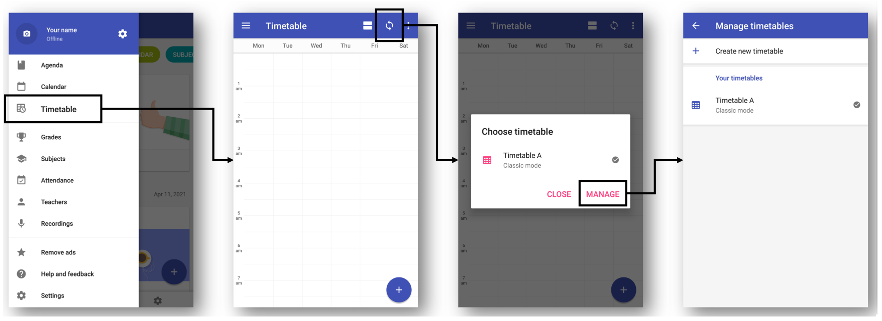
Listing 6.1: Test script for timetable management in School Planner

For example, Listing 6.1 shows a GUI test for a feature dealing with timetable management in School Planner, a popular planner app for students [31]. The execution of this test case is depicted in Figure 6.1-a. This test resembles the actions a user would take to manage timetables through a dedicated pop-up menu. The assertion in the last line checks if the app responds correctly by verifying the title of the page is "Manage timetables" (the last screen of Figure 6.1-a). Nevertheless, Figures 6.1-b and 6.1-c demonstrate that this feature can be actually performed in two alternative ways, i.e., via the timetable selection dialog or Settings. When the developer writes the test for this feature, she may perceive the scenario of Figure 6.1-a as the default or primary way of invoking this functionality. Failure to create tests for the other two ways of invoking the feature, however, leaves the potential faults that can only be revealed by those tests undetected. For instance, the first test shown in Figure 6.1-a cannot reveal latent faults in the event listener method of the "MANAGE" button in Figure 6.1-b.

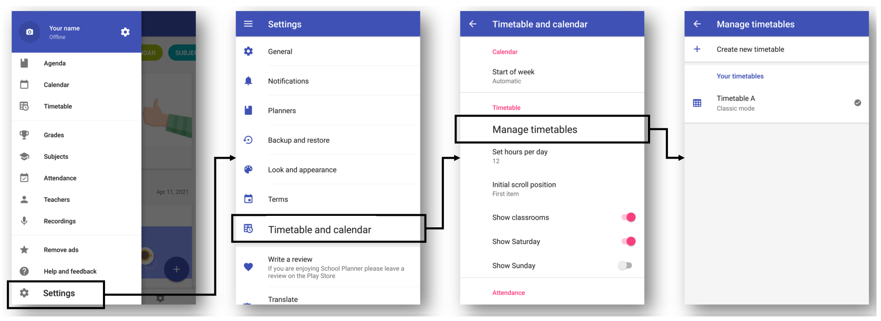
To reduce the manual effort of creating feature-based tests and help developers more thoroughly verify the features of their apps, we present ROUTE, short for *ROads not taken in Ui TEsting*, which is an automated solution for feature-based UI test augmentation for Android apps. ROUTE first takes a feature-based UI test, including both its inputs and assertions, and the AUT as input. It then applies novel heuristics to explore the AUT and generate additional UI tests that verify *the same feature* as the original test. ROUTE leverages virtualization techniques to increase the accuracy and performance of app exploration. In other words, it saves and restores the snapshots of the memory of the device in certain states. In fact, Figures 6.1-b and 6.1-c are the alternative scenarios discovered by ROUTE from the



(a) Origianl scenario: manage timetables through a dedicated pop-up menu



(b) Alternatie scenario: manage timetables through the timetable selection dialog



(c) Alternatie scenario: manage timetables through Settings

Figure 6.1: Different use-case scenarios for timetable management in the School Planner app.

original test shown in Figure 6.1-a.

ROUTE has several key differences from prior work in test augmentation [111, 102, 129, 71, 125, 46, 65, 39, 121, 80, 132, 33]. First, the proposed augmentation is feature-based. In other words, we aim to generate tests that verify the same functionalities as the original tests. To achieve this goal, we have developed several properties that the generated tests should hold and designed ROUTE based on these properties. Second, the assertions in the original tests are reused in the generated tests, which makes the augmented test suites capable of detecting feature-related faults and providing the developers with actionable information to debug their programs. Finally, we have developed a test generation algorithm that prioritizes the candidate tests according to how likely they are to exercise a feature in a materially different way than the existing tests.

We have applied Route on several dozens of feature-based UI tests for verification of popular apps on Google Play. The experimental results show that for 89% of the existing tests, Route was able to generate at least one alternative test. Moreover, the fault detection effectiveness of the augmented test suites was improved by up to 39% over the original test suites.

Overall, this chapter makes the following contributions:

- We propose a feature-based UI test augmentation technique capable of creating high-quality tests, consisting of both inputs and assertions, to verify features of an app in alternative ways.
- We present novel heuristics for the proposed augmentation, and implement them as an automated tool, called Route, which is publicly available [19].
- We empirically evaluate Route on real-world tests and demonstrate its utility to generate additional tests capable of detecting new faults.

The rest of this chapter is organized as follows. Section 6.2 elaborates on the concept of feature-based test augmentation. Section 6.3 provides an overview of Route as well as the implementation details of its components. Section 6.4 presents our test generation algorithm. Section 6.5 discusses the evaluation results. The chapter concludes with future work.

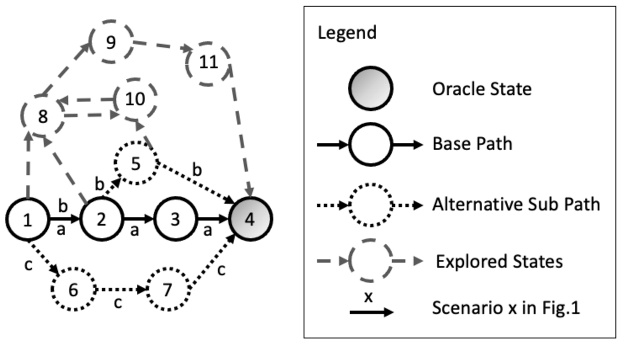


Figure 6.2: Example of states and paths in the School Planner app shown in Figure 6.1

6.2 Background

An existing UI test can be augmented by modifying its execution path. In this work, we model the dynamic behavior of the AUT as a graph $G = (V, E)$, where:

- V is a set of GUI states (screens). Each $v \in V$ represents a unique runtime GUI state in the AUT.
- E is a set of edges between the GUI states. Each $e = (v_i, v_j, (w, a)) \in E$ represents a transition from v_i to v_j by firing a GUI event that performs an action a on a widget w.

Furthermore, the execution of a test can be perceived as a traversal through the graph. Specifically, for a UI test t, we represent its execution path $p = (v_s, v_f, V_p, E_p, V_o)$ as follows:

- $V_p \in V$ are GUI states visited by t with edges $E_p \in E$.
- $v_s \in V_p$ denotes the start state, i.e., the state before t executes the first event.
- $v_f \in V_p$ denotes the end state, i.e., the state after t executes the last event. v_s and v_f are also referred as *terminal states*.
- $V_o \subseteq V_p$ denotes the *oracle states*, i.e., the states on which t has assertions. We also store the assertions associated with each $v_o \in V_o$.

Finally, for a usage-based test to be augmented, we call its execution path *base path* and the visited GUI states *base states*. For example, in Figure 6.2, the solid edges illustrate the base path of the test executing the scenario of Figure 6.1-a. The base path can then be modified to generate new tests from the original test.

When we construct a modified execution path p' from a base path p to generate a new test, we would like p' to still test the same functionality as p, albeit using an alternative path. To that end, we propose several properties that we would like the modified execution path p' to hold. First, the start and end states of p' should be the same as the base path p, because the terminal states provide important information about the boundary of the tested feature. Similarly, the oracle states V_o in p, as well as the assertions examined at each state $v_o \in V_o$, should also be included in p', because they are critical for semantically verifying the functionality. Finally, p' should not be drastically different from p. The reason is that overly modifying p may significantly change the behavior of the AUT and invalidate the original assertions. As a result, we need to limit the changes from p to p' to specific types of operations, such as replacing a sub-path in p with an alternative simple path. The paths labeled "b" and "c" in Figure 6.2 (corresponding to the scenarios of Figures 6.1-b and 6.1-c) exemplify the modified execution paths satisfying the proposed properties. In the following sections, we describe how we designed ROUTE to integrate these properties into the generated tests.

6.3 Approach

Figure 6.3 provides an overview of ROUTE. It takes an original test and the AUT as input and generates new tests that examine the same feature as the original test in alternative ways. There are three main components in ROUTE: *Base Path Construction*, *App Exploration*, and *Test Generation*. First, Base Path Construction component executes the original test and

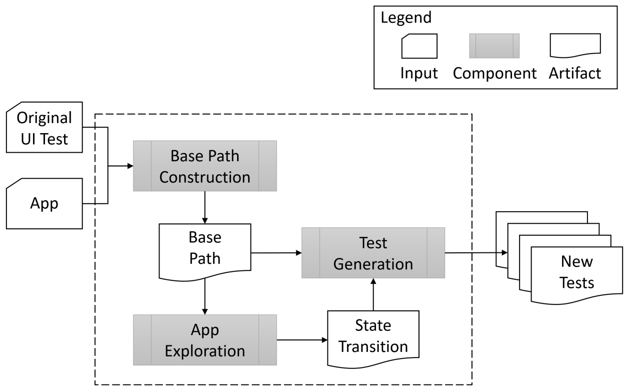


Figure 6.3: Overview of ROUTE

retrieves its execution path as the base path. Next, based on the visited states in the base path, App Exploration component systematically explores the AUT and outputs the state transition diagram that represents the AUT's runtime behavior. Finally, Test Generation component applies novel heuristics to identify and prioritize the executable tests that hold the desired properties discussed in Section 6.2. We describe the implementation of each component in the following subsections.

6.3.1 Base Path Construction

Algorithm 5 describes how Base Path Construction component executes the original test t to obtain its execution path p as the base path. As defined in Section 6.2, p is a 5-tuple of v_s (start state), v_f (end state), V_p (base states), E_p (edges), and V_o (oracle states). The algorithm first initializes E and V_o as an empty set, and then launches the AUT. Next, it dumps the current screen of the AUT to obtain the initial state before test execution as the start state, and use it to initialize base states and the variable for previous state (line 3-6). A dumped screen of an Android app is a widget hierarchy tree in XML format, in which non-leaf nodes are layout widgets and leaf nodes are actionable or visible widgets, such as buttons and text views. We uniquely identify a GUI state by computing a hash value over the widget hierarchy tree.

Algorithm 5 ROUTE: Base Path Construction

```
Input:
    $t$: original test
Output:
    $p = (v_s, v_f, V_p, E_p, V_o)$: the base path of $t$

1: $E_p = \emptyset$; $V_o = \emptyset$                ▷ initialize edges and oracle states
2: $launchApp()$
3: $screen = dumpCurScreen()$
4: $v_s = getHash(screen)$                             ▷ start state
5: $V_p = \{v_s\}$                                     ▷ initialize base states
6: $prevState = v_s$
7: $takeSnapshot(v_s)$
8: for each $event \in t$ do
9:     if $event$ is not an assertion then
10:        $execute(event)$
11:        $screen = dumpCurScreen()$
12:        $curState = getHash(screen)$
13:        $V_p = V_p \cup curState$
14:        $E_p = E_p \cup (prevState, curState, event)$
15:        $prevState = curState$
16:        $takeSnapshot(curState)$
17:    else                                            ▷ $event$ is an assertion
18:        $V_o = V_o \cup prevState$
19:    end if
20: end for
21: $v_f = prevState$                                  ▷ end state
22: $p = (v_s, v_f, V_p, E_p, V_o)$
23: return $p$
```

Before executing the test, a snapshot of the start state is taken (line 7). In this work, we leverage virtualization to save the visited states and later restore and explore them in App Exploration (detailed in the next subsection). In other words, the AUT is installed and executed on a virtual machine (VM) such as Android Virtual Device [27] or VirtualBox VM [109], such that the runtime program state of the AUT, including the underlying OS and emulated hardware, can be stored in a snapshot and fully resumed later. This helps improve the accuracy and performance of GUI state exploration. Typically in prior work [1, 70, 95, 97, 123], a GUI state is resumed by restarting the AUT and replaying the recorded event sequence. However, a shortcoming of this restart-and-replay approach is that the background services may change after restarting the AUT and the GUI state cannot be reached again. Virtualization addresses this issue. Moreover, the exploration of the AUT can be accelerated because the snapshots can be restored and processed in parallel with multiple VMs. The practice of using virtualization and snapshots is also adopted by prior work in Android testing [57].

Algorithm 5 executes the original test t after the initialization steps (line 8). For each $event \in t$, if the event is not an assertion, it first executes the event and then obtains *curState*, the current GUI state after execution (lines 9-12). We subsequently update the base path in terms of the base states V_p and edges E_p, and take a snapshot of this new state (line 13-16). On the other hand, if the event is an assertion[1], it means some checks are performed on the previous state, and we hence update the oracle states V_o with *prevState* (line 17-18). Finally, after the execution of t, the end state is updated and the base path p is returned as input to the next phase of App Exploration (line 21 to 23).

6.3.2 App Exploration

With the base path p from the original test, App Exploration component performs *k-step lookahead* on each base node to obtain G, an explored state transition graph of the AUT, as described in Algorithm 6. In particular, the algorithm explores if there are paths with length not greater than k from a base state v to another base state. This base-path-directed exploration restricts the possible paths that can be constructed from the graph later, such that the generated tests will not deviate too much from the base path.

Algorithm 6 first initializes G, the graph to be returned, with the states and edges in the base path (line 1). Next for each base state $v \in V_p$, it performs initialization steps in lines 3-8. It creates a first-in-first-out *queue* to store the event sequences that need to be executed for exploration (line 3). To initialize the queue, it retrieves all actionable widgets from v

[1]The assertions considered in this work are UI-based assertions, such as checking the existence of widget or text, or verifying the attributes of a widget.

Algorithm 6 ROUTE: App Exploration

```
Input:
    $p = (v_s, v_f, V_p, E_p, V_o)$: base path
    $k$: lookahead threshold
Output:
    $G$: state transition diagram of the AUT

 1: $G = \{V_p, E_p\}$
 2: for each $v \in V_p$ do
 3:     $queue = \emptyset$                                   ▷ A first-in-first-out queue
 4:     $widgetList = getActionable(v)$
 5:     for each $widget \in widgetList$ do
 6:         $events = \emptyset$
 7:         $events = events \cup (widget, getAction(widget))$
 8:         $queue.push(events)$
 9:     end for
10:     while $queue \neq \emptyset$ do
11:         $restoreSnapshot(v)$
12:         $events = queue.pop()$
13:         $subPath = \emptyset$
14:         $prevState = v$; $curState = v$
15:         for each $e \in events$ do
16:             $execute(e)$
17:             $screen = dumpCurScreen()$
18:             $curState = getHash(screen)$
19:             if $curState \neq prevState$ then
20:                 $subPath =$
                        $subPath \cup (prevState, curState, e)$
21:                 $prevState = curState$
22:             end if
23:         end for
24:         if $curState \in V_p$ then
25:             $G.update(subPath)$
26:         else if $events.length < k$ then
27:             $widgetList = getActionable(curState)$
28:             for each $widget \in widgetList$ do
29:                 $newEvents =$
                        $events \cup (widget, getAction(widget))$
30:                 $queue.push(newEvents)$
31:             end for
32:         end if
33:     end while
34: end for
35: return $G$
```

(line 4), creates an action event, such as *click*, for each of them, and pushes the generated event sequences into *queue* (lines 5-8). For example, the solid nodes and edges in Figure 6.2 depict the initial G for the test in Listing 6.1. Moreover, for v equal to state 2, *queue* is initialized with three single events, which when executed result in exploration of states 3, 5, and 8.

For each event sequence $events \in queue$, Algorithm 6 first restores the GUI state v and executes the events sequentially (line 11-16). The executed events, as well as the encountered GUI states, are also recorded as *subPath*, a sub-path starting from v (lines 17-21). Next, if *curState*, the GUI state reached after the execution, is a base state, it updates G with the traversed *subPath* (line 24-25). Otherwise, if the length of *events* is smaller than the threshold k, it means we can continue exploring forward from v. To that end, we retrieve all actionable widgets for *curState* and create *newEvents*, a new event sequence by appending an appropriate action event, such as *click* or *type*, to each of the retrieved widgets (line 27-29). *newEvents* is then pushed into *queue* for further exploration (line 30). For example, for $k = 2$ and v equal to state 2 in Figure 6.2, subpaths $(2 \rightarrow 3)$ and $(2 \rightarrow 5 \rightarrow 4)$ will be added to G since states 3 and 4 belong to V_p. Note that, some paths (explored states in Figure 6.2) are not added to G because they do not visit any base states, or they exceed the lookahead threshold, e.g., given $k = 3$, the path $(1 \rightarrow 8 \rightarrow 9 \rightarrow 11 \rightarrow 4)$ is not explored. After all event sequences in *queue* are consumed for all base states v, the algorithm stops and returns G (line 35). We describe Test Generation in detail in the next section.

6.4 Test Generation

Test Generation component takes the base path p and the state transition diagram G as input, and generates T, a set of executable tests with size n, as described in Algorithm 7. To that end, in line 2, it retrieves all *simple paths* in G that (1) are from start state (v_s) to

Algorithm 7 ROUTE: Test Generation

```
Input:
    $p = (v_s, v_f, V_p, E_p, V_o)$: base path
    $G$: state transition diagram of the AUT
    $n$: number of tests to be generated
Output:
    $T$: a set of executable tests with size $n$

 1: $T = \emptyset$
 2: $paths = getSimplePaths(G, v_s, v_f, V_o)$
 3: $T = generateTestsFromPaths(T, paths, V_o)$
 4: if $|T| < n$ then
 5:     $paths = getCyclicPaths(G, v_s, v_f, V_o)$
 6:     $T = generateTestsFromPaths(T, paths, V_o)$
 7: end if
 8: return $T$

 9: function GENERATETESTSFROMPATHS($T, paths, V_o$)
10:     $paths = sort(paths)$
11:     for each $p' \in paths$ do
12:         $launchApp()$
13:         $executable = execute(p', V_o)$
14:         if $executable$ is $true$ then
15:             $T = T \cup p'$
16:             if $|T| = n$ then
17:                 return $T$
18:             end if
19:         end if
20:     end for
21:     return $T$
22: end function
```

end state (v_f), and (2) visit all oracle states (V_o). We focus on the paths that satisfy these two constraints because they are required properties for the generated tests (as discussed in Section 6.2). Moreover, tests with an execution path containing a cycle may include repetitive operations such as navigating back to a previous screen, and hence considered less useful. As a result, the algorithm first considers simple paths (i.e., paths without repeating states) when generating the tests.

Next, Algorithm 7 verifies the retrieved simple paths and generates executable tests from them (line 3) by calling the `generateTestsFromPaths()` (line 9). This function first sorts

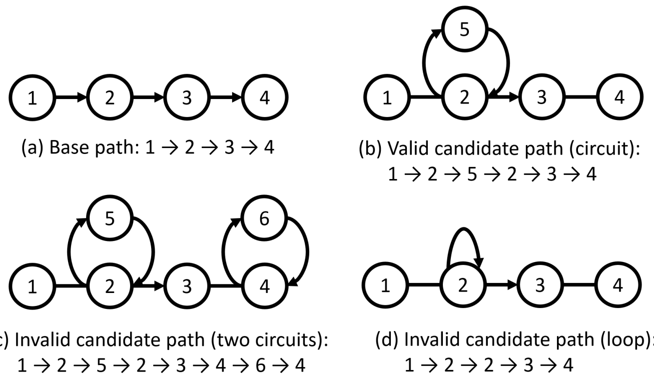

(a) Base path: 1 → 2 → 3 → 4
(b) Valid candidate path (circuit): 1 → 2 → 5 → 2 → 3 → 4
(c) Invalid candidate path (two circuits): 1 → 2 → 5 → 2 → 3 → 4 → 6 → 4
(d) Invalid candidate path (loop): 1 → 2 → 2 → 3 → 4

Figure 6.4: Examples of valid and invalid paths considered by `getCyclicPaths()` in Algorithm 7

the paths by their difference from the base path (line 10, detailed in the next subsection), and then launches the AUT and executes each of the prioritized paths to verify if it is executable (line 11-13). If a candidate path p' is executable, it is added to T (line 14-15). Note that when verifying p', if an oracle state $v_o \in V_o$ is encountered, the associated assertions are also performed. That way, the generated tests include the original assertions that have passed. `generateTestsFromPaths()` returns when the size of T is n (line 16) or all candidate paths are verified (line 21).

If the size of T is smaller than n after checking all the simple paths, Algorithm 7 will further retrieve the execution paths containing only one cycle (line 4-5). Specifically, `getCyclicPaths()` in line 5 only considers the paths with a non-loop circuit (i.e., a cycle with a length larger than one) on a base state, as illustrated in Figure 6.4. We only allow this special case for execution paths containing cycles, because we would like to modify the base path conservatively. The retrieved paths are then verified in the same way to generate more tests (line 6). The algorithm returns when the required number of tests are generated or after checking all valid candidate paths (line 8).

Algorithm 8 Route: Path Difference

```
1: function pathDiff($p, p'$)
2:     $L = getStateList(p)$                          ▷ $L = \{v_1, v_2, ..., v_{|L|}\}$
3:     $L' = getStateList(p')$                        ▷ $L' = \{v'_1, v'_2, ..., v'_{|L'|}\}$
4:     $d_1 = editDist(L, L')$
5:     $score = 0;\ count = 0$
6:     for $i$ from 2 to $|L'|$ do
7:         if $v'_i \notin L$ then
8:             $score = score + xmlEditDist(v'_i, v'_{i-1})$
9:             $count = count + 1$
10:        end if
11:    end for
12:    $d_2 = score/count$
13:    return $\alpha \times d_1 + (1 - \alpha) \times d_2$
14: end function
```

6.4.1 Computing path difference

With the explored state transition graph of the AUT, the number of candidate paths may grow exponentially and become too many to verify. As a result, Test Generation component prioritizes the candidate execution paths by their difference from the base path (line 10 in Algorithm 7). In other words, we compute a distance score between the base path p and each candidate path p' with the `pathDiff()` function in Algorithm 8. The paths with the highest distance scores are executed and verified first.

We would like to select the candidate paths that are most different from the base path, as they are more likely to exercise a feature in a materially different way than the existing test. To that end, we consider the difference between a candidate path and the base path at both the *path level* and the *state level*. By path level, we mean how different the two paths are in terms of their length and visited states. At the state level, we further look into the new states in the candidate path and determine the extent they are different from their preceding states.

Algorithm 8 first gets the list of states visited by p and p' as L and L', respectively (lines 2-3). It then computes a coarse-grained score (d_1) about how different L and L' are, in terms of their edit distance (line 4). Here we treat L and L' as two strings and the contained state identifiers (hash values) as characters, and compute the number of required edits, inserts or deletions that converts L to L'. Next, it traverses each state $v'_i \in L'$ sequentially. If v'_i is not a base state (i.e., $v'_i \notin L$), we compute a find-grained score about how different v'_i and v'_{i-1} (its previous state) are, in terms of the edit distance between their XML representations (line 8). This helps us estimate if the change from v'_{i-1} to v'_i is huge or merely incremental (such as a checkbox is selected). Finally, d_1 and the averaged find-grained score (d_2) are normalized into $[0, 1]$ and a weighted sum of them is returned (line 13, with $\alpha = 0.5$ in our implementation).

6.5 Evaluation

We investigated the following research questions in our experimental evaluation of Route:

RQ1. What types of tests are generated by Route? Are they for the same feature and meaningful?

RQ2. Is the fault detection effectiveness of the augmented test suites improved?

RQ3. How is the fault detection effectiveness of augmented test suites affected by the independent variables in Route?

6.5.1 Experimental Setup

We implemented Route with Python for UI tests written using Appium [8], an open-source and cross-platform testing framework. Existing usage-based tests for the subject apps are written with Appiums' Python client. The generated tests are stored in JSON format and

Table 6.1: Subject tests and experimental results

TestName	AppName	#Events	#Assertions	#Alt Tests Generated	%Mutants Killed	
					Orig	Augmented
TestAddFavorite	Astro	5	1	3/3	2	2
TestAppManager	Astro	2	1	0/3	5	16
TestCreateFolder	Astro	5	1	3/3	2	2
TestListView	Astro	3	1	2/3	7	11
TestOpenDirectory	Astro	3	1	3/3	2	2
TestSearch	Astro	6	1	2/3	9	9
TestCreateLink	OwnCloud	10	3	3/3	11	13
TestDelete	OwnCloud	6	2	3/3	3	13
TestFileDetail	OwnCloud	9	2	2/3	3	8
TestRename	OwnCloud	7	1	3/3	8	18
TestSearch	OwnCloud	3	1	1/3	0	11
TestSearchDetail	OwnCloud	4	1	2/3	0	5
TestUpload	OwnCloud	5	1	2/3	0	0
TestViewStorage	OwnCloud	2	1	1/3	0	11
TestAddGrade	School Planner	8	1	3/3	8	14
TestAddSubject	School Planner	8	1	3/3	0	14
TestAgenda	School Planner	3	1	3/3	0	6
TestCalendar	School Planner	2	1	3/3	0	6
TestManageTimeTable	School Planner	5	1	3/3	6	11
TestSettings	School Planner	3	1	3/3	0	6
TestViewProgression	School Planner	3	1	3/3	0	8
TestViewTimeTable	School Planner	4	1	3/3	6	8
TestAddNewItem	Shopping List	4	1	3/3	0	0
TestAddNewList	Shopping List	5	1	2/3	0	10
TestCheckAll	Shopping List	3	1	2/3	0	13
TestDeleteChecked	Shopping List	6	1	2/3	3	15
TestDeleteList	Shopping List	3	1	1/3	3	13
TestRenameList	Shopping List	5	1	0/3	3	10
TestSearch	Shopping List	4	1	2/3	0	13
TestAddDraft	WordPress	10	1	3/3	7	25
TestBlogPost	WordPress	9	1	3/3	4	21
TestMenuBlog	WordPress	2	1	2/3	4	21
TestMenuMedia	WordPress	2	1	2/3	0	18
TestMenuPage	WordPress	2	1	2/3	0	18
TestPageSearch	WordPress	4	1	2/3	0	25
TestPostNavigation	WordPress	9	4	3/3	4	21
TestViewSite	WordPress	2	1	2/3	0	21
TestCreateFolder	Writely Pro	5	1	1/3	0	17
TestCreateNote	Writely Pro	6	1	3/3	0	39
TestDeleteNote	Writely Pro	4	1	3/3	0	3
TestEditNote	Writely Pro	4	1	3/3	0	25
TestMoveNote	Writely Pro	6	1	0/3	3	17
TestRenameNote	Writely Pro	5	1	0/3	3	6
TestSearch	Writely Pro	3	1	0/3	3	28
Total		209	51	95/132	12	33

executed by our test runner. In our experiments, we used VirtualBox VM [109] with Android-x86 7.1 OS [23] installed. The experiments were conducted on a Windows laptop with 2.8 GHz Intel Core i7 CPU and 32 GB RAM. Our experimental data is publicly available [19].

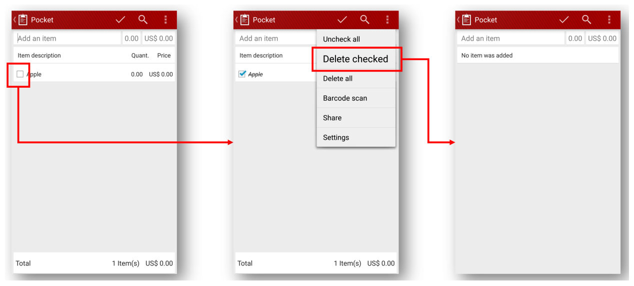
Subject tests. We collected a set of feature-based UI tests from prior work in mobile app testing [42, 119]. In a few cases, we added appropriate assertions to the tests that had no assertion. Table 6.1 shows the subject tests used in our study, including the test names, app names, and the contained GUI events and assertions. In total, 44 original feature-based tests were included. Five of the apps under test (except for Writely Pro) are popular apps, with 100K+ to 50M+ installs on Google Play.

Independent variables. The two primary independent variables in ROUTE are k, the lookahead step in App Exploration, and n, the number of generated tests in Test Generation. We also investigated how these two variables influence the augmented test suites in RQ3.

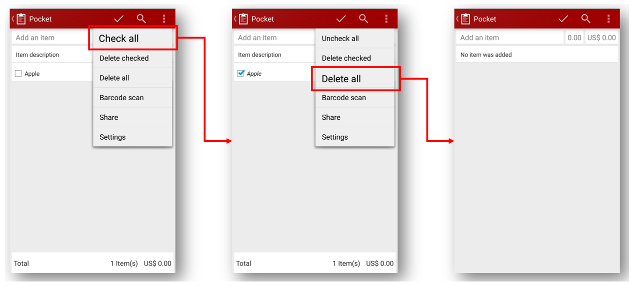
6.5.2 RQ1: Quality of the Generated Tests

In our experiment, 132 new tests were generated by ROUTE (with $n = 3$) from the 44 original tests in Table 6.1. We first manually classified all the generated tests as *alternative tests* or not. A generated test is an alternative test if its execution path contains an acyclic sub-path from one base state to another base state that is different from the sub-path taken by the original test. Table 6.1 shows that 72% (95/132) of the generated tests are valid alternative tests for verifying a feature. For 89% (39/44) of the original tests, ROUTE was able to generate at least one alternative test. Furthermore, in most cases (90%, 35/39) the alternative tests are the first test generated by ROUTE. This indicates that the prioritization criteria of ROUTE works effectively and it can quickly find alternative tests.

We also conducted a qualitative study by manually examining the behavior of the 95 alternative tests generated by ROUTE and categorized them into four groups.



(a) Original: click the checkbox and delete checked items via a menu option



(b) Generated: select and delete all items via menu options

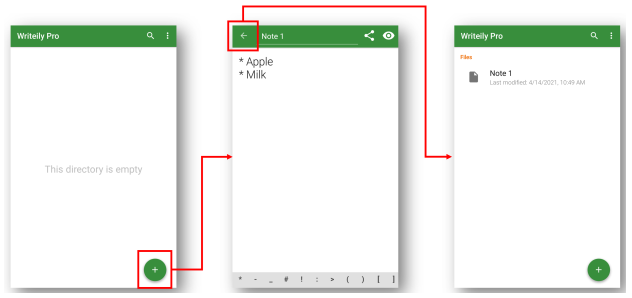
Figure 6.5: Deleting a checked item in Shopping List with different controllers

Different Controllers

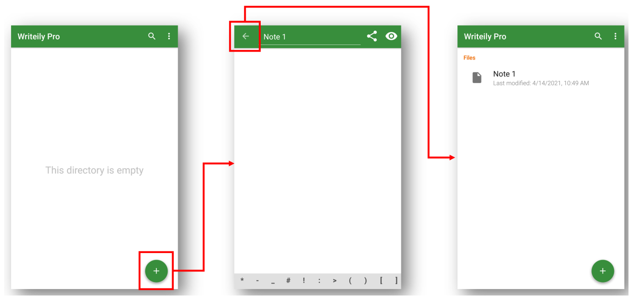
In this category, the generated test performs the same task through different GUI controllers. Examples include Figure 6.1-b and 6.1-c. Instead of the pop-up menu in the original test, these two tests reach the timetable management screen by the selection dialog and menu options in Settings, respectively. Another example is illustrated in Figure 6.5. In the original test, an item is selected by clicking the checkbox and then deleted through a menu option. This task can be instead finished by different controllers, i.e., menu options that select and delete items, as performed in the generated test.

Different Input Data

In this category, the generated test performs the same task through the same controllers, but with different input data. For example, Figure 6.6 illustrates that the task of creating



(a) Original: create a note with title and content



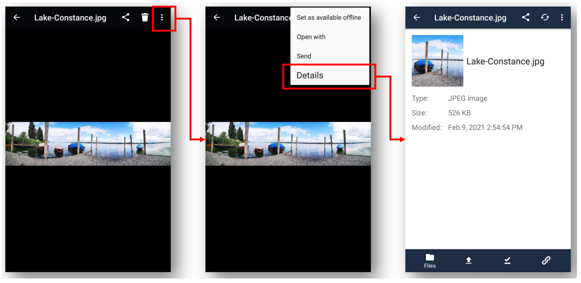
(b) Generated: create a note with title only

Figure 6.6: Creating a note in Writeily Pro with different input data

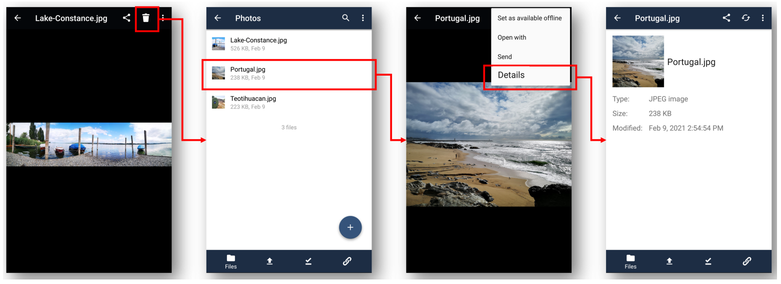
a note is tested the same way in both the original and generated tests, i.e., using the same controllers. However, the created notes are different in terms of the content (i.e., input data). Figure 6.7 provides another example in this category. An original test depicted in Figure 6.7-a views the details of a photo, but in the generated test the photo is deleted and the details of another photo is displayed (Figure 6.7-b). In other words, the task of viewing photo details is perform the same way, i.e., via the same menu option, in these two tests, but the photos (input data) are different.

Different Control Flow

The generated test performs the same task through the same controllers and with identical input data. However, it adds additional steps to the original control flow. For instance, Figure 6.8-a shows that the generated test switches from table view to list view before managing timetables. Similarly, Figure 6.8-b illustrates that an additional circuit is added



(a) Original: view the details of a photo via a menu option



(b) Generated: delete the photo and then view the details of anoter photo, via the same menu option

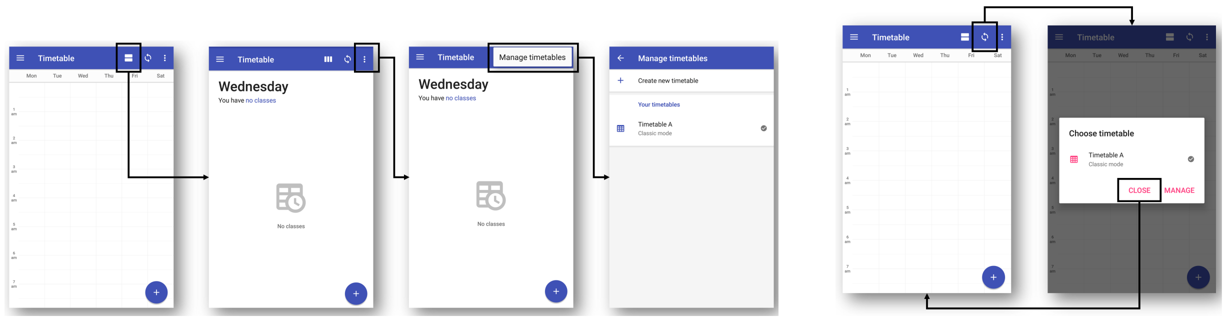
Figure 6.7: View photo details in OwnCloud with different input data

to the original control flow by opening and closing a dialog.

Deviated

Here, the original assertions accidentally pass in the generated test, but the desired functionality is not actually performed. For example, an original test for the page search feature in WordPress first performs a search with a keyword and then checks if a specific page title is displayed. On the other hand, one of the generated tests directly navigates to the page list screen and performs the existence check of the page title. Such a test is not considered to be performing the same task as the original test.

The number of generated tests for each category is listed in Table 6.2. We believe the first three categories of tests can be useful to developers in different ways, while the deviated tests may not be. 93% (88/95) of the alternative tests generated by ROUTE fall in the first



(a) Switch from table view to list view before managing timetables

(b) Open and close a dialog before managing timetables

Figure 6.8: Timetable management in School Planner with different control flows

Table 6.2: Categories of tests generated by ROUTE

Category	#Generated Tests
Different Controllers	38 (40%)
Different Input Data	17 (18%)
Different Control Flow	33 (35%)
Deviated	7 (7%)
Total	95 (100%)

three categories.

6.5.3 RQ2: Fault Detection Effectiveness

To investigate whether the generated tests are advantageous in terms of fault detection effectiveness, we created mutants for the apps under test with MutApk [60], an open-source mutation testing tool for APK files. It supports 35 mutation operators designed for Android apps [88] and performs the mutation on intermediate representations of the code. We created the mutants with MutApk's default strategy, in which both the mutation operators and locations of mutated code were picked randomly. The number of mutants created for each app is shown in Table 6.3.

The percentage of mutants killed by the original test suite and the augmented test suite is shown in Table 6.1. Note that we generated 3 more tests for each original test, so the size of augmented test sets is four (one original plus three generated). Table 6.1 shows that

Table 6.3: Mutants created for each subject apps

Subject App	#Mutants
Astro	44
OwnCloud	38
School Planner	36
Shopping List	39
WordPress	28
Writely Pro	36
Total	221

Table 6.4: Percentage of mutants killed by the original tests and augmented tests with different lookahead step k and number of generated tests n

original	n=3			k=3		
	k=1	k=2	k=3	n=1	n=2	n=3
12	13	25	33	29	32	33

the fault detection effectiveness of the augmented test sets improved on 86% (38/44) of the subject tests and by up to 39% (in the case of *TestCreateNote* for Writely Pro). In total, the augmented tests were able to kill 33% (74/221) of the mutants, while the original tests were only able to kill 12% (27/221) of them. This indicates that the generated tests are not redundant. They covered additional parts of the AUT that the original tests missed. In other words, ROUTE is capable of augmenting test suites to detect latent faults.

6.5.4 RQ3: Influence of the Independent Variables

In this research question, we investigated how the lookahead step k and the number of generated tests n influence the fault detection effectiveness (FDE) of augmented suites. To that end, we first fixed $n = 3$ and checked the effect of k. In other words, for the 44 original tests we produced three sets of augmented tests with exactly the same size (i.e., $44 + 132 = 176$). The difference among them is that they are generated according to state transition diagrams resulting from different values of k. We report the percentage of mutants killed by each of these three sets by considering all of the generated mutants. As shown in Table 6.4, the FDE of the augmented test suites improves as k increases. This indicates that the ability to look further when exploring the AUT is important for ROUTE to produce tests with high FDE.

Next, we fixed $k = 3$ and checked the effect caused by n. That is, we used the same state transition graph from 3-step lookahead to produce three sets of augmented tests with different size, in which the first set is one time larger than the original tests ($n = 1$), the second set is two times larger ($n = 2$) and so on. The results in Table 6.4 indicate that, generally the first test generated by ROUTE is very good in terms of FDE. For example, by generating one more test for each original test, the FDE was improved from 12% to 29%. However, ROUTE can still improve the FDE of augmented suite by adding more tests.

6.6 Conclusion

There are often several ways of invoking the core features of an app. Due to the manual effort of writing tests, developers tend to consider only the typical way of invoking a feature when creating the tests. However, the alternative ways of invoking a feature are as likely to be faulty, which would go undetected without proper tests. This chapter presented ROUTE, an automated tool for feature-based UI test augmentation for Android apps. ROUTE creates high-quality tests, consisting of both inputs and assertions, to verify the features tested by existing tests in alternative ways. ROUTE relies on several novel heuristics to guide the search for new tests and leverages virtualization technology to save the visited UI states, such that the states can be fully restored later for exploration.

Experimental results using real-world subjects have demonstrated the effectiveness of ROUTE, as it successfully generated alternative tests for 89% of the existing test cases in our experiments. Moreover, the fault detection effectiveness of augmented test suites in our experiments showed substantial improvements of up to 39% over the original test suites.

In our future work, we aim to investigate the applicability of techniques described here in other computing domains (e.g., web applications) and other testing paradigms (e.g., unit tests). We also plan to conduct user studies with practitioners to validate the utility of tests generated using ROUTE in practice.

Chapter 7

Test Automation in Open-Source Android Apps: A Large-Scale Empirical Study

Automated testing of mobile apps has received significant attention in the recent years from researchers and practitioners alike. In this chapter, we report on the largest empirical study to date, aimed at understanding the test automation culture prevalent among mobile app developers. We systematically examined more than 3.5 million repositories on GitHub and identified more than $12,000$ non-trivial and real-world Android apps. We then analyzed these non-trivial apps to investigate (1) the trends in adoption of test automation; (2) working habits of mobile app developers in regards to automated testing; and (3) the correlation between the adoption of test automation and the popularity of projects. Among others, we found that (1) only 8% of the mobile app development projects leverage automated testing practices; (2) developers tend to follow the same test automation practices across projects; and (3) popular projects, measured in terms of the number of contributors, stars, and forks on GitHub, are more likely to adopt test automation practices. To understand the rationale behind our observations, we further conducted a survey with 148 professional and experienced developers contributing to the subject apps. Our findings shed light on the current practices and future research directions pertaining to test automation for mobile app development.

7.1 Introduction

Testing is an indispensable phase of software development life cycle. It is the primary way through which quality of software is improved. In comparison with manual testing, automated testing is reported to be more advantageous for a number of reasons, such as reliability, repeatability, and execution speed, especially in the context of continuous integration [58]. To understand the test automation culture prevalent among mobile app developers, researchers have investigated the extent to which test automation is adopted in practice [81, 90, 54, 55, 77]. However, those studies are limited in terms of both scale and quality of the curated dataset.

First, most prior works have only considered hundreds of apps from a single source, i.e., F-Droid. The findings and conclusions drawn from a relatively small set of sample apps may not generalize to the overall app ecosystem.

Second, previous studies have failed to exclude dummy and invalid tests; an important factor that might severely affect their conclusion. That is, when developers create a new project with Android Studio, the official IDE for Android app development, it generates some example test cases which are irrelevant for the created app. Including these default tests may influence the results of research questions as to the adoption of test automation practices.

Finally, appropriate and representative subjects are of critical importance for an empirical study. In the case of test automation for Android apps, a practical inclusion criterion is to consider only *non-trivial* apps, since it is not cost-effective to write tests for trivial apps such as class assignments, tutorials, or simple apps with only one component. Studying trivial apps cannot reveal useful insights into the adoption of test automation practices. Nevertheless, no previous study has focused exclusively on non-trivial apps.

In this chapter, we report on a large-scale empirical study on open-source Android apps from GitHub. We systematically examined more than 3.5 million non-forked repositories in Java and Kotlin, and investigated more than $12,000$ real-world apps to determine (1) the trends in adoption of test automation; (2) working habits of mobile app developers in regards to automated testing; and (3) the correlation between the adoption of test automation and the popularity of projects in terms of different metrics, such as contributors and stars on GitHub, and ratings on Google Play Store.

Two important contributions of our work are the scale of study and the way we have curated the dataset. First, we considered more than $12,000$ apps across 16 app markets including Google Play Store, F-Droid, and PlayDrone. We also developed novel heuristics to exclude irrelevant and example tests in data collection and analysis. Lastly, the subject apps were selected according to a criteria designated for identifying non-trivial apps (detailed in Section 7.3). As presented in Section 7.4, these efforts led to findings that are quite different from prior work.

Another contribution of our work is that we considered both unit tests and UI tests. Given the interactive nature of mobile apps, UI testing, which requires an emulator or a real device to run, is the primary way to examine the functionality and usability of mobile apps. Therefore, in addition to unit tests, we are interested in whether and how automated UI tests are adopted by mobile developers. We discuss related research questions such as developers' preference for unit and UI testing and their compliance with the Testing Pyramid practice [68] in Section 7.4.

To gather a deeper understanding of the underlying reasons for our observations from the source code, we further conducted a survey with the contributors of the subject apps, and ended up with 148 responses mainly from professional and experienced developers. Interestingly, with respect to some of the research questions, the results obtained from the analysis of project data and survey responses are inconsistent, indicating a gap between what the developers believe they do versus what they actually do.

Overall, this chapter makes the following contributions:

- We report on the first large-scale analysis focusing on non-trivial apps in over $12,000$ open-source projects from 16 app markets and spanning a period of 5 years, to investigate how test automation is practically adopted.
- We present the working habits of mobile app developers regarding test automation, such as the tendency to write tests or lack thereof and the compliance with the Testing Pyramid practice.
- We discuss how the presence of automated tests, and its extent, impact the popularity of apps in terms of different metrics on GitHub and Google Play Store.
- We present the findings of a survey involving 148 practitioners who developed the subject apps to understand the rationale behind our observations as well as the challenges in Android app testing.
- We create a publicly available dataset for this study [30]. The dataset was built by referring to multiple data sources including GitHub, Google Play Store, F-Droid, and AndroZoo.

The remainder of this chapter is organized as follows. We provide a background on mobile app test automation in Section 7.2. In Section 7.3, we present our approach for data collection, subject selection, and developer survey. In Section 7.4, we present and discuss our findings. The chapter concludes with a discussion of threats to validity and future work.

7.2 Test Automation in Android

7.2.1 Unit and UI Tests

Given the interactive nature of mobile apps, there are roughly two types of tests in Android: unit tests and UI tests.[1] According to the definition from Google [68], unit tests are small tests that "validate the app's behavior one class at a time". In contrast, UI tests or end-to-end tests are medium or large tests that "validate user journeys spanning multiple modules of the app". The key difference between unit and UI tests, besides the scope of testing, is that unit tests run on a local machine with JVM, while UI tests need an emulated or real device to run, and almost always use the Android OS or Android framework.

In Android Studio, the official IDE for Android app development, unit and UI tests are clearly separated—they are placed in different directories. The tests in the *test* folder are unit tests that run locally on JVM. The tests in the *androidTest* folder are UI tests that require an emulator or real device to run. These two directories are automatically generated when developers create a new project with Android Studio. In this study, we consider the tests under the *test* folder as unit tests, and the tests under the *androidTest* folder as UI tests.

A feature of Android Studio highly related to our study is that, when developers create a new project, it generates not only the folders, but also examples for different types of tests. By default, the *test* folder contains a class called *ExampleUnitTest.java*, and the *androidTest* folder contains a class called *ExampleInstrumentedTest.java*, as shown in Figure 7.1. They are executable examples of unit and UI tests to help developers get started with test automation. However, including these example files may result in overestimated conclusions for research questions about the prevalence or adoption of automated tests, because developers

[1]Sometimes they are called local tests and instrumented tests [67].

```
public class ExampleInstrumentedTest {
@Test
public void useAppContext() {
Context appContext = InstrumentationRegistry
.getInstrumentation()
.getTargetContext();
assertEquals("com.example",
appContext.getPackageName());
}
}

public class ExampleUnitTest {
@Test
public void addition_isCorrect() {
assertEquals(4, 2 + 2);
}
}
```

Figure 7.1: Example Test Classes Generated by Android Studio

may accidentally commit these files without an intention to write automated tests. In this study, we exclude these example files when counting number of tests contained in an app.

7.2.2 The Testing Pyramid Practice

The Testing Pyramid is a mindset or practice to guide developers in terms of how much effort they should put on creating different kinds of automated tests [53, 64, 68, 32, 107]. It essentially says that developers have to balance their automated tests by having many more low-level unit tests than high-level UI tests, as illustrated in Figure 7.2.

There are many reasons to follow the Test Pyramid practice. First, unit tests make debugging easier because they focus on small modules that can be tested independently. When unit tests fail, developers can quickly pinpoint the root cause of failure and save a lot of time. On the other hand, if there is a failure reported by a UI test, it usually means that the corresponding unit tests are incorrect or missing. Furthermore, unit tests are more robust and run faster in general, while UI tests may be subject to flakiness [99] and almost always

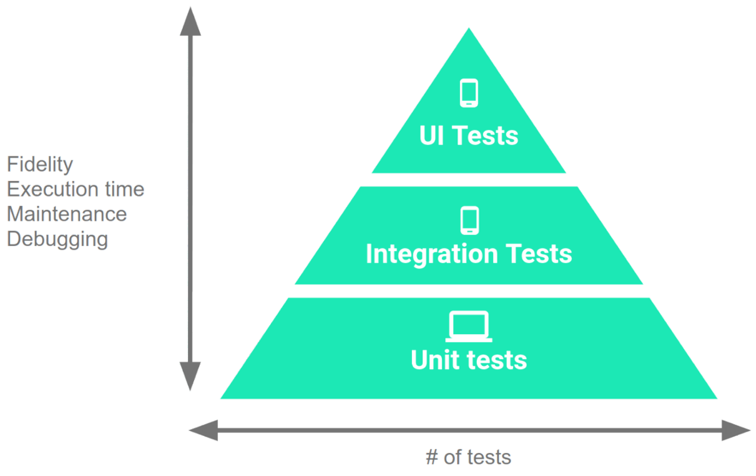


Figure 7.2: Illustration of the Testing Pyramid Practice from [68]

run slower. As a result, while UI tests are still important to validate end-to-end workflows, overly relying on them will make testing expensive, slow, and brittle.

Although the proportion of tests for each layer in the Testing Pyramid varies based on different apps, a general recommendation from Google is a 70/20/10 split: 70% unit tests, 20% integration tests, and 10% UI tests [68]. Note that, while there is a layer of integration tests, and they can be understood as tests that "validate the collaboration and interaction of a group of units [68]", the scope for integration tests is controversial [79]. In fact, these three layers are not totally clear-cut and sometimes overlap with each other [108]. In this chapter, we leverage the characteristics of Android apps and Android Studio to identify the two major types of tests, unit and UI tests. Furthermore, according to the above guideline, we believe an appropriate ratio of UI tests should be 20% to 30% of the total number of tests.

7.3 Methodology

Figure 7.3 depicts the flow of data collection and analysis in our study. This study consisted of the following steps: (1) we first collected a large list of GitHub repositories from the

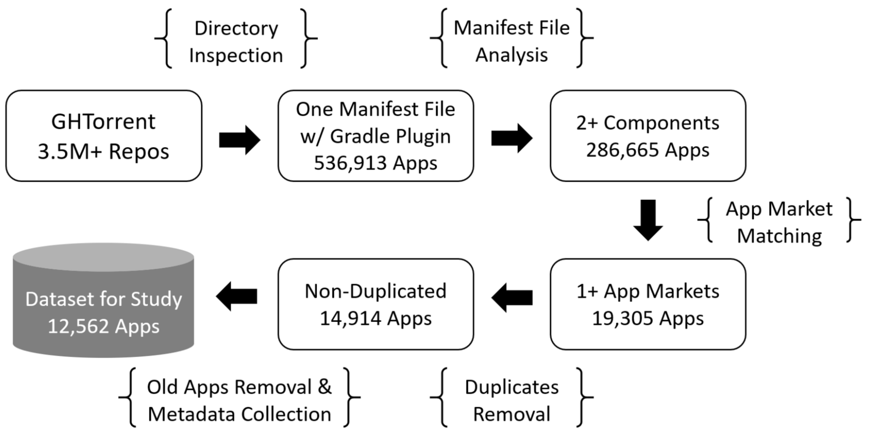

Directory Inspection
Manifest File Analysis
GHTorrent 3.5M+ Repos
One Manifest File w/ Gradle Plugin 536,913 Apps
2+ Components 286,665 Apps
App Market Matching
1+ App Markets 19,305 Apps
Non-Duplicated 14,914 Apps
Dataset for Study 12,562 Apps
Old Apps Removal & Metadata Collection
Duplicates Removal

Figure 7.3: Flow of Data Collection and Analysis in This Study

GHTorrent database [69]; (2) we set filtering criteria to identify the repositories representing non-trivial Android apps; (3) we further analyzed the identified repositories to collect their meta-data and information about automated tests and popularity; (4) we evaluated the collected dataset to answer research questions about the test automation culture prevalent among mobile app developers; and finally (5) we conducted a survey with the developers of the subject apps to get a deeper understanding of the underlying reasons for our observations from the dataset. We now describe each of these steps in further detail.

7.3.1 Study Subjects and Selection Criteria

The initial list of GitHub repositories for our study was queried from the GHTorrent database [69], a research project that monitors the GitHub public event time line and populates the collected information with a relational database. We downloaded the latest dump of their database [66], and queried the repositories written in Java or Kotlin that are neither forked nor deleted. The query returned with a list of more than 3.5 million repositories.

To identify the repositories of non-trivial and real-world Android apps from the returned list, we set the following selection criteria:

(1) The repository must contain exactly one *AndroidManifest.xml*. The manifest file is a must-have for every Android app to provide essential information about the app to the Android build tools [25]. The reason for exactly one manifest file is that the repository containing multiple such files is likely a tutorial or class assignment with multiple demo apps. We used GitHub API to walk through the directory tree of the projects to search for the files.

(2) The repository must contain `build.gradle` with a specific string `‘‘com.android.application’’` inside. Android Studio uses Gradle as its build system, and a Gradle plugin with this specific string means that this project has a task to build an Android app. We used GitHub API to search the projects with the specified condition. Most of the repositories were filtered out with these two criteria, with about 537 thousand apps left.

(3) At least two components have to be declared in the manifest file. We parsed the manifest file and looked for the declaration of four Android component types (i.e., Activity, Service, Broadcast Receiver, and Content Provider [24]) inside. We set a threshold of 2 components because we believe it is not cost-effective to write tests for a simple app with only one component. About half of the apps were removed by this step, with 287 thousand apps left.

(4) The package name stated in the manifest file must appear in an app market. We believe that the apps published in app markets, especially the markets that charge fees to join such as Google Play Store, are more likely beyond toy or demo apps, because the developers want the apps to reach general users (and even willing to pay for it). From the manifest file of each app, we retrieved the package name and tried to match it with apps hosted in the following app markets: Google Play Store, F-Droid [61], and the list of package names and markets provided by AndroZoo [34].[2] This criterion was critical to identify non-trivial apps and left us with a list of about 19 thousand apps.

[2]A list of app markets considered by AndroZoo can be found at [38].

(5) We removed the apps with duplicate package names, and ended up with a list of 14,914 GitHub repositories of non-trivial Android apps.[3]

The above filtering process took two months, primarily because of the rate limit of GitHub API (5,000 requests per hour).

7.3.2 Data Collection and Analysis

For each of the selected repositories, we used GitHub API to further collect its meta-data: creation date, number of forks, number of stars, number of commits, number of contributors, number of issues, and number of pull requests. If the app is on Google Play Store, we also collected its category and user ratings by crawling the app page.

To collect the information about how test automation is adopted in the project, we used GitHub API to walk through the directory tree of the project, and parsed all the files under the *test* and *androidTest* folders, if any exist. We considered a method as a test case if it is annotated with "@Test". This annotation is used by JUnit-based testing frameworks, including both unit and UI testing frameworks such as JUnit [78], Robolectric [115], Mockito [105], and Espresso [20]. A prior study investigating the usage of testing frameworks in 1,000 apps on F-Droid [55] shows that 100% of the adopted unit testing frameworks and 97% of the UI testing frameworks are JUnit-based. Furthermore, we classify a test case as a unit test if it is under the *test* folder, and otherwise as a UI test (i.e., under the *androidTest* folder). Finally, as mentioned in Section 7.2.1, we excluded the example unit and UI test generated by Android Studio.

An assumption of our study is that the subject apps were developed with Android Studio. Because Android Studio has been the official IDE for Android app development since its first

[3]Sometimes two repositories contain the same package name because one is a direct copy of the other (not by forking). In this situation, we keep the repository with the oldest creation date.

Table 7.1: Distribution of Apps by App Market*

Market*	#Apps
Google Play	11265
PlayDrone	539
fdroid	434
anzhi	408
appchina	294
mi.com	70
VirusShare	62
angeeks	41
1mobile	26
freewarelovers	12
slideme	10
torrents	4
praguard	3
hiapk	2
proandroid	1
apk_bang	1

*An app may belong to multiple markets

Table 7.2: Distribution of Apps by Year Created

Year Created	#Apps
2015	3614
2016	2330
2017	1731
2018	2898
2019	1989
Total	12562

stable release in December 2014 [26], we further factored out the repositories before 2015 from the list described in Section 7.3.1. We finally ended up with $12,562$ repositories/apps in our dataset. The distribution of apps by app market is shown in Table 7.1. While the majority of the apps were published on Google Play Store, the dataset covers apps across 16 app markets. Table 7.2 shows the distribution of apps by the year they were created. For the apps on Google Play Store, Figure 7.4 shows the distribution by category.

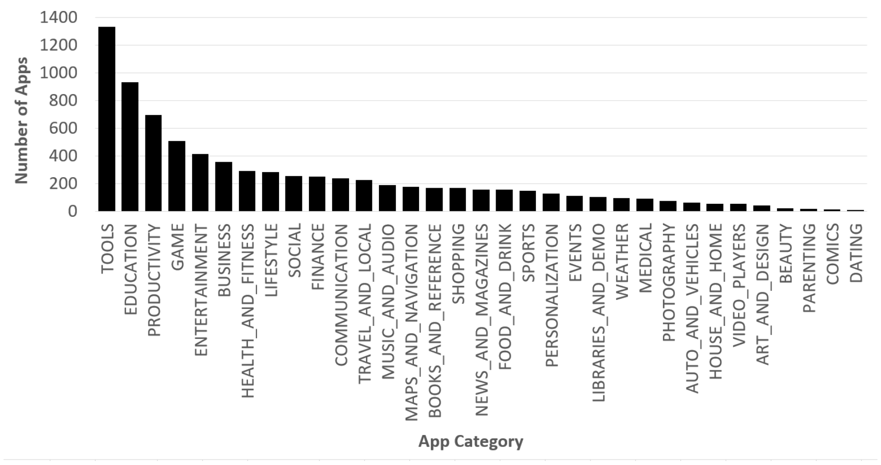


Figure 7.4: Distribution of the Google Play Apps by Category

7.3.3 Survey

To complement our findings, we conducted an online survey with the developers of the subject apps in our dataset. In this section, we describe the design, participant selection, and data collection of the survey.

Survey Design

The online survey was designed to understand the rationale behind our findings from the dataset as well as the challenges in Android app testing. We first asked demographic questions to understand the respondents' background, such as their experiences in terms of the number of years of Android app development. We then asked them about their current practices of Android app testing. For the respondents reporting the use of automated tests, we further asked them related questions such as the preference for unit and UI testing and whether they follow the Testing Pyramid practice, and the reasons for their choices. Next, we presented some of our findings in the correlation analysis between the adoption of test automation and the popularity of apps, and asked for their opinions on possible explanations. Finally, we asked the respondents for the difficulties in adopting automated tests and general challenges of testing Android apps. For all questions about practices and opinions, we provided a set of choices identified from previous studies [90, 81, 55], as well as an "other" choice with free form text if none of the provided choices apply. A sample of the survey can be found at the companion website [30].

To ensure that the questions were clear and the survey can be finished in 10 minutes, we conducted a pilot survey with graduate students in Computer Science who have experience in Android app development and survey design. We rephrased some questions according to the feedback. The responses from the pilot survey were used solely to improve the questions and were not included in the final results.

Participant Selection

From each subject app in our dataset, we tried to retrieve the email of its main contributor in the following order: (1) the email found in the GitHub profile of the repository's owner; (2) the email of the contributor who made the most commits; and (3) the email of the contributor who made the most recent commit. After removing invalid and duplicate data, we identified $7,490$ unique email addresses for our survey.

Data Collection

We used Qualtrics [113] to distribute the survey to the $7,490$ targeted email addresses, and 653 of them bounced. From the $6,837$ emails successfully sent, we received 148 valid and complete responses with a 2.2% response rate. The response rate is close to the results of previous studies such as 2.1% (83/3905) reported in [81] and 1.0% (102/10000) reported in [90] on very similar surveys with mass developers on GitHub.

The 148 received responses are from 45 countries. The top two countries where the re-

Table 7.3: Distribution of Apps in Terms of Presence of Test Cases

Group	#Apps	Percentage
Apps with any tests	1002	7.98%
Apps without tests	11560	92.02%
Apps with unit tests	766	6.10%
Apps with UI tests	502	4.00%
Apps with both unit and UI tests	266	2.22%

spondents reside are United States of America (22.3%) and India (10.8%). 70.3% of the respondents are professional software developers paid by a company, and 68.9% of them have more than 2 years of experience in Android app development.

7.4 Results

In this section, we present the results of our study from the following perspectives: (1) the prevalence and trends in the adoption of test automation; (2) working habits of the mobile app developers with respect to test automation; and (3) the correlation between the adoption of test automation and the popularity of projects.

7.4.1 Prevalence and Trends in the Adoption of Test Automation

To understand the degree to which test automation is adopted in open-source Android app development, we investigate the following research questions:

RQ1. How prevalent is test automation in open-source Android apps, in terms of the presence of unit and UI tests?

Table 7.3 shows the number of repositories grouped by the presence of different types of tests. The results indicate that only 7.98% of the subject apps contain tests, and most of them are poorly tested in an automated manner—even though they are non-trivial. This percentage is much lower than previous findings: 20% reported in [54], 14% reported in [81], and 40% reported in [55].

There are many possible reasons for the inconsistency between our results and previous findings. First, our analysis excludes the placeholder tests that are automatically generated by Android Studio, as mentioned in Section 7.2.1. This check was critical for our results, since such tests are common in our dataset (7,017 of the 12,562 apps examined, 56%). We also manually checked the dataset released by Coppola et al. [54], and found such examples in the reported test cases. We are not able to verify the results reported by Kochhar et al. [81] because they are not willing to release their dataset. Regarding the results reported by Cruz et al. [55], since they did not search for test cases (detailed in the next paragraph), we are unable to compare their results with ours.

The way one computes the existence of tests can also influence the results significantly. For example, in the study by Cruz et al. [55], they inspect the build configurations and look for imports related to testing frameworks to determine the presence of tests in a repository. Since having related imports in the build configurations does not necessarily mean there are test cases in the project, their findings about prevalence of test automation is prone to overestimation.

Finally, the scale of study might also affect the results. In the papers by Kochhar et al. [81] and Cruz et al. [55], only 627 and 1,000 apps from F-Droid were analyzed, respectively. In contrast, our study considers more than 12,000 apps on GitHub across 16 markets, which is substantially different from their works in terms of scale and source of data.

Another finding from Table 7.3 is that UI testing is not adopted as extensively as unit testing (i.e., 4% vs. 6.1%). We will further discuss this in Section 7.4.2.

Table 7.4: The Ways of Testing Android Apps by the Survey Participants

Way	#Respondents
Manually	130
With scripted/automated tests	85
With dedicated QA team or 3rd party testing services	43
With automatic input generation tools	12
Other	6
Not at all	3

Table 7.5: The Reasons for not Adopting Test Automation by the Survey Participants

Difficulty	#Respondents
Cost to create and maintain automated tests	77
Time constraints	74
Size or maturity of the app	66
Lack of exposure or knowledge of existing frameworks	52
Cumbersome to use	50
Lack of support from management or organization	30
Other	11

Observation 1: Only 8% of the non-trivial and real-world apps have automated tests. Automated UI testing is less adopted than unit testing.

RQ2. How prevalent is test automation and what are the reasons for not adopting it (as reported by developers)? What are the challenges in testing of Android apps in general?

In our survey, we asked the developers how they test their Android apps, and they were allowed to select all options that apply. Table 7.4 shows the results. Interestingly, over 57% (85/148) of the respondents state that they are using automated tests, yet we do not observe this degree of test automation adoption from the subject apps they develop. One possible explanation for this inconsistency is that the proponents of test automation are more willing to take our survey, while the developers not interested in test automation have no incentive to provide feedback. Another reason could be that the professional developers adopt automated tests at work, but not for their pet projects on GitHub. Finally, it is also possible that the developers only uploaded their source code on GitHub without corresponding tests.

To understand why the observed adoption of test automation is low, we asked the developers

Table 7.6: The Biggest Challengs in Testing Android Apps by the Survey Participants

Challenge	#Respondents
Fragmentation	104
Concurrency	66
Performance	51
Security	44
Energy	43
Functionality	43
Accessibility	35
Other	14

Table 7.7: The Most Important or Useful Criteria for Evaluating Android App Tests by the Survey Participants

Criterion	#Respondents
Fault detection capability of tests	96
Feature or use case coverage of tests	83
Code coverage of tests	67
Code or test case reviews	59
Other	7

to specify the reasons for not adopting test automation. From the results in Table 7.5, we can see the top three reasons are: (1) cost to create and maintain automated tests, e.g., caused by changing requirements or rapid development; (2) time constraints, e.g., because of time-to-market or customer's schedule; and (3) size or maturity of the app, e.g., the app is not big or complex enough to require automated tests. Note that the third reason corresponds to our insight that it is not cost-effective to write automated tests for trivial apps, and they should be excluded in the empirical study, as we have done. Besides, the respondents also mentioned other interesting difficulties in adopting test automation as follows:

"Legacy code not designed to be tested requires lots of refactoring which makes it harder to justify the additional effort to write tests."

"...hard to test unexpected GUI aspects or unexpected hardware (manufactor firmware) issues or unexpected permission issues or unexpected Android behavoir or unexpected 3rd party data formats."

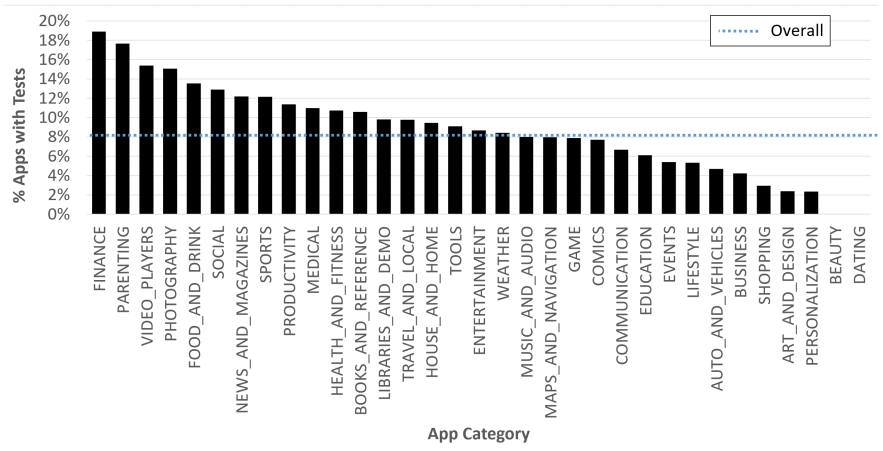


Figure 7.5: Prevalence of Test Automation of the Google Play Apps by Category

It is worth mentioning that we also asked two general questions to understand (1) the biggest challenges in testing of Android apps; and (2) the most useful criteria for evaluating tests for Android apps. The results are reported in Tables 7.6 and 7.7. According to Table 7.6, the top three challenges are: (1) fragmentation, e.g., multiple Android OS or API versions, devices with different sizes or resolutions, etc.; (2) concurrency, e.g., detecting data races, deadlock, or violation of execution order of methods; and (3) performance, e.g., app's responsiveness such as frames per second for gaming apps. Moreover, from Table 7.7 we can see that the developers do not consider code coverage as the most important criterion for evaluating tests, which is in line with the prior study [90]. We believe the reported concerns call for additional research and development in test automation frameworks and tools.

Observation 2: 57% of the survey participants reported the use of test automation, which varies drastically from that observed in the dataset. The top three difficulties in adopting test automation are: cost to create and maintain tests, time constraints, and size or maturity of the app.

RQ3. Is the prevalence of test automation varied across different categories of apps?

To understand whether there are any patterns as to the adoption of automated testing prac-

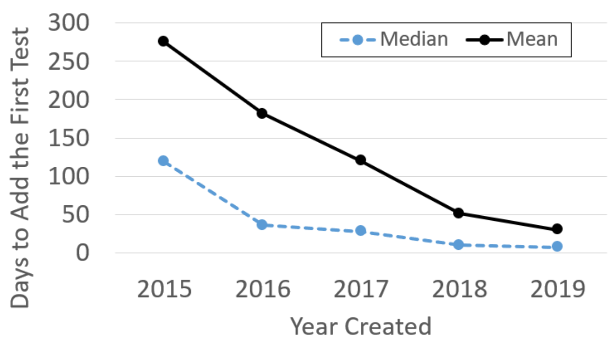


Figure 7.6: Days to Add the First Test after Project Creation

tices across different categories of apps, for the Google Play apps with category information in our dataset, we report their adoption of automated tests by category in Figure 7.5. As depicted in Figure 7.5, while overall the prevalence of test automation is 8%, the percentage is substantially higher for some categories of apps such as finance (19%) and video players (15%). On the other hand, some categories of apps such as shopping (3%) and dating (0%) are poorly tested in an automatic manner. This variance could be attributed to the quality requirements for different categories. Note that the observed patterns may not generally apply to apps on Google Play Store, since many commercial and closed-source apps, such as popular shopping apps, are not included in our study.

Observation 3: Some categories of apps, such as finance and video players, are more extensively leveraging test automation techniques than others.

RQ4. How long does it take for a project to adopt test automation?

It is not rare that testing is conducted or introduced later in mobile app development. To understand how long it takes for the first automated test to be added to a project, we analyzed the commit logs of all 1,002 apps with tests in our dataset, and computed the time interval in days for each app between the creation of the project and the first automated test. We report the medians and means of the results by the creation year of the apps in Figure 7.6. From Figure 7.6 we can see a clear trend that the adoption of automated tests is

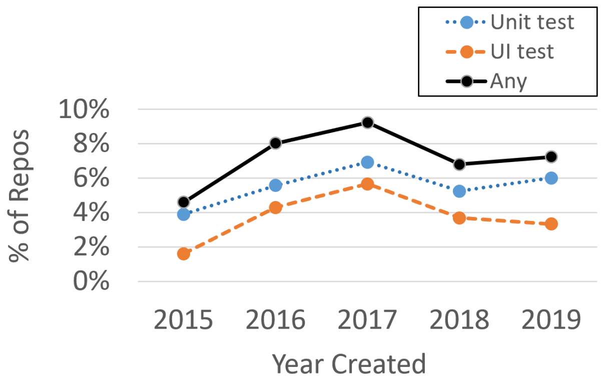


Figure 7.7: Percentage of Repositories Containing Tests over the Past 5 Years

faster in more recent apps. For instance, the apps created in 2015 took 275 days on average to add the first unit or UI test, and half of the apps took more than 119 day to do so (the median). On the other hand, the apps created in 2019 took only 30 days on average to add their first test, with a median of 7 days.

Observation 4: Developers that are willing to adopt test automation, adopt it in their projects more quickly than before. The average number of days to adoption drops from 275 days in 2015 to 30 days in 2019.

RQ5. Has the adoption of test automation increased over time?

Tools and libraries for test automation in Android are continuously evolving. While we are far from an extensive adoption of test automation, we would like to know if it is gaining momentum within the mobile app developer community or not. Therefore, we analyzed how test automation is used in the apps created in the past 5 years. Given that old apps may have more time than new apps to add tests, to compare them fairly, we checked the use of test automation within 9 months after project creation for all apps. The threshold of 9 months is selected based on the answer to previous research question regarding the average days to adoption of test automation.

Figure 7.7 shows the percentage of the apps containing tests grouped by the year they are created. While Figure 7.7 illustrates a rising trend of the adoption of test automation from 2015 to 2017, it does not appear to be the case from 2017 to 2019. For example, the percentage of repositories containing any type of tests increases from 4.6% in 2015 to 9.2% in 2017, but then goes down to 7.2% in 2019. This observed pattern applies to both unit and UI testing. In other words, we do not see a trend that test automation has become more popular in recent years. Note that our finding is inconsistent with the study by Cruz et al. [55], in which they observed newer apps have more tests than older apps. We believe the reasons for this are similar to what we have discussed in the first research question (i.e., the fact that they did not properly curate their dataset).

Observation 5: Test automation adoption does not appear to be gaining momentum among the open-source Android projects. We do not observe a trend that test automation has become more popular in recent years.

7.4.2 Working Habits of Mobile App Developers

RQ6. Do the same developers have the same testing habits across apps?

In this section, we investigate whether developers are following the same test automation habits across apps. To that end, we first clustered all subject apps by their owner, i.e., the GitHub account, and obtained a set of 985 clusters, S_a, in which each cluster contains two or more apps by the same developer. Next, we defined and computed the *test adoption rate* for each cluster C in S_a as follows:

$$rate(C) = \frac{\#Apps\ with\ test\ in\ C}{\#Apps\ in\ C}$$

A cluster with a rate of 1 or 0 means the developer has followed the same behavior across apps. That is, the developer either wrote tests for all of her apps or did not write tests at

Table 7.8: Probability of Observing Consistent Behavior on the Apps by the Same Developers. S_a: Clusters of Apps by the Same Developers. S_b: by Different Developers

Set	Size (#Clusters)	#Clusters Same Behavior	Probability	p-value
S_a	985	902	91.57%	4.06e−12
S_b	985	763	77.46%	

all. We further computed the probability of observing the same behavior in S_a by dividing the number of clusters showing the same behavior (i.e., achieve test adoption rate of 1 or 0) by the size of S_a.

Moreover, to understand if the probability observed in S_a is high, we created another set of clusters, S_b, as a control group. The number of clusters and the size of each cluster in S_b is exactly the same as S_a. However, the apps in S_b were randomly selected from the apps not in S_a. We computed the test adoption rate for each cluster in S_b using the same equation, and the probability of observing the same behavior in S_b accordingly.

Finally, to determine if the observed difference between S_a and S_b is statistically significant, we applied hypothesis testing on the rate distribution of S_a and S_b using the non-parametric test Mann-Whitney U [96] with a significance level of 0.05. We chose the Mann-Whitney U test because S_a and S_b are not normally distributed and did not pass the normality test of Shapiro-Wilk [122].

The results in Table 7.8 show that in S_a, the set of clusters in which each cluster consists of the apps by the same developer, it is more likely to observe a cluster manifesting the same behavior. In other words, for a group of apps by the same developer, the probability that either all or none of them have tests (91.57%) is higher than a group of apps by different developers (77.46%). The difference between S_a and S_b is statistically significant, because the null hypothesis that S_a and S_b are from the same distribution is rejected by the Mann-Whitney U test with a *p-value* of 4.06×10^{-12}.

Table 7.9: The Reasons for the Preference of Unit Testing by the Survey Participants

Reason	#Respondents
Speed	39
Scope	30
Simpleness	28
Robustness	26
Other	6

Observation 6: App developers tend to follow the same test automation practices across projects.

RQ7. Do developers prefer unit or UI testing and why?

From Table 7.3 in Section 7.4.1, we see that the apps adopting unit tests (6.1%) are more than UI tests (4%). To validate our observation and understand the reasons behind this, for the developers reporting the use of test automation, we further asked what type of testing (unit testing or UI testing) they do mostly and why. Among the 83 respondents, the majority of them (55/83, 66%) prefer unit testing. This is in line with our observation from the dataset. Furthermore, 27% (22/83) of the respondents have no preference and 7% (6/83) of them prefer UI testing.

We also asked the proponents of unit testing for their rationale. Table 7.9 shows that the top three reasons by the developers are: (1) speed, e.g., unit tests run faster than UI or end-to-end tests; (2) scope, e.g., unit tests focus on small or independent modules, thereby simplify the debugging; and (3) simpleness, e.g., unit tests are easier to learn and write. On the other hand, developers preferring UI testing indicate that the interactivity is the top reason, because UI or end-to-end tests can test the app in a more interactive and straightforward way.

Observation 7: Majority of the developers prefer unit testing, corroborated through both project dataset and survey results. The top three reasons are speed, scope and simpleness.

Table 7.10: Distribution of the Number of Tests in the Apps with Both Types of Tests. 1Q: 1st quartile. 2Q: 2nd quartile (median). 3Q: 3rd quartile.

	Distribution					
	Min	Max	Mean	1Q	2Q	3Q
#Unit Tests	1	685	34.75	3	11	27.25
#UI Tests	1	178	14.32	2	7	17
Ratio of UI Tests to All Tests	0.3%	97.1%	41.9%	17.5%	40.0%	64.4%

RQ8. Is the practice of Test Pyramid followed by developers?

As mentioned in Section 7.2.2, the Testing Pyramid practice is a guideline for developers to have a balanced portfolio of different types of automated tests. To understand if the guideline is appropriately followed by the developers, we analyzed the 266 apps containing both unit tests and UI tests in our dataset by counting the number different types of tests. Furthermore, we computed the ratio of the number of UI tests to the total number of tests as follows:

$$\frac{\#UI\ tests}{\#Unit\ tests + \#UI\ tests} \times 100$$

Table 7.10 shows that the distribution of the numbers of unit and UI tests in the apps are skewed, because the averages are much larger than the medians (i.e., 34.75 vs. 11 for unit tests, and 14.32 vs. 7 for UI tests). That means some apps contain many more tests than others. On the other hand, the third quartile shows that 75% of the apps have fewer than 27.25 unit tests and 17 UI tests. We believe these are reasonable numbers for general apps.

Regarding the developers' compliance with the Test Pyramid practice, Table 7.10 shows that in more than half of the apps, the ratio of UI tests is higher than 40%, which differs from the recommended ratio of 20%-30% by Google [68]. In other words, the developers put more effort than recommended in writing UI tests. A possible explanation is that the interactive nature of mobile apps drives the developers to write more UI tests. However, as mentioned

Table 7.11: Distribution of the Popularity and Satisfaction Metrics of the Apps. 1Q: 1st quartile. 2Q: 2nd quartile (median). 3Q: 3rd quartile.

	Distribution						
	Sample Size	Min	Max	Mean	1Q	2Q	3Q
Stars	12533	0	7897	15.09	0	0	1
Forks	12533	0	2209	4.49	0	0	1
Contributors	12533	0	451	2.30	1	1	2
Commits	12527	1	13844	75.95	4	15	55
Issues	12527	0	2442	5.5	0	0	0
Pull Requests	12527	0	1679	3.80	0	0	0
Ratings	3937	1	5	4.23	3.91	4.38	4.78

in Section 7.2.2, overly relying on UI tests may make testing and debugging cumbersome. As a result, while UI tests are essential to validate certain types of requirements such as business logic and usability, we believe that the developers need to better follow the Test Pyramid practice and write more unit tests.

In our survey, we asked the participants whether they are following the Testing Pyramid practice, and 52% (51/98) of them said no, which is consistent with our observation from the dataset. A prominent reason from the respondents reporting the non-compliance is the lack of exposure or knowledge about the Testing Pyramid practice (40/51, 78%). Other interesting reasons include "special needs for my team or projects" and "the Testing Pyramid practice is misleading/flawed".

Observation 8: Developers put more effort than recommended in writing UI tests, as the average ratio of UI tests to all tests is 40%.

7.4.3 Association Between Presence of Automated Tests and Popularity

Mobile app developers often strive to have their apps become popular. On the one hand, as members of an open-source community, developers are happy to see their apps getting more attention from other developers in terms of stars, forks, contributors, etc. on GitHub.

On the other hand, as product owners, developers want their apps to satisfy the users and get good ratings and feedback on the market. While these popularity metrics are not necessarily related to the development process of apps, we would like to investigate whether they are impacted by the adoption of test automation. Specifically, we consider the following popularity metrics on GitHub: number of stars, forks, contributors, commits, issues[4], and pull requests. Moreover, we consider user ratings on Google Play Store as the metric of user satisfaction. These metrics were collected in the manner described in Section 7.3.2. Table 7.11 presents the distribution of data in terms of different metrics.

RQ9. How does test automation relate to project popularity?

We would like to know whether apps with tests are different from apps without tests in terms of the popularity metrics on GitHub. First, to eliminate the effect caused be app size, we excluded the apps that have fewer than 3 components (the 1st quartile) and more than 8 components (the 3rd quartile) in our dataset, ending up with a set of $7,664$ apps under consideration. Next, we conducted statistical analysis for each metric with the following steps:

(1) We divided the data into two disjoint sets, R_w and R'. R_w consists of the metric values from the apps with tests. R' consist of the metric values form the apps without tests.

(2) We applied the Z-score method [83] with a threshold of three times of standard deviation to remove the outliers from both sets.

(3) Since the apps without tests are much more than the apps with tests in our dataset, R_w and R' are extremely unbalanced in terms of their sizes. Given that unequal sample sizes may generally reduce statistical power of equivalence tests [118], we created R_o with the same size as R_w by randomly selecting the values in R'.

[4]Issues may be considered as an indicator of app quality. In fact, the topics posted with issues can be very broad, such as feature request or usage discussion. Therefore, we consider it as an indicator of popularity.

Table 7.12: Impact of Having Tests on the Popularity of Apps. R_w: Apps with Tests. R_o: Apps Without Tests.

	Stars*				Forks*			
	Size	Mean	Median	p-value	Size	Mean	Median	p-value
R_w	629	10.95	0	1.96e−10	630	3.74	0	1.83e−11
R_o	629	3.34	0		630	1.61	0	
Δ		7.61	0			2.13	0	

	Contributors*				Commits*			
	Size	Mean	Median	p-value	Size	Mean	Median	p-value
R_w	630	2.76	2	9.88e−17	628	147.21	84.5	4.70e−70
R_o	630	1.55	1		628	38.07	14	
Δ		1.21	1			109.14	70.5	

	Issues*				Pull Requests*			
	Size	Mean	Median	p-value	Size	Mean	Median	p-value
R_w	635	10.39	0	4.74e−30	633	8.76	0	2.27e−32
R_o	635	1.29	0		633	0.87	0	
Δ		9.1	0			7.89	0	

*The difference is statistically significant.

(4) We computed the mean and median of R_w and R_o and the difference between the mean and median.

(5) To determine if the difference observed in R_w and R_o is statistically significant, as in Section 7.4.2, we performed hypothesis testing on R_w and R_o using the Mann-Whitney U test with a significance level of 0.05. The null hypothesis on R_w and R_o is that they were selected from populations having the same distribution. For example, in the case of stars, the null hypothesis is that "an app with tests (from R_w) has the same number of stars on GitHub as an app without tests (from R_o)". We chose the Mann-Whitney U test because R_w and R_o are not normally distributed and did not pass the normality test of Shapiro-Wilk.

(6) The above process is repeated for all the popularity metrics.

Table 7.12 shows the results of our statistical analysis. The statistical evidence shows that test automation is associated with all popularity metrics. Namely, on average, open-source Android apps with tests are expected to have more stars, forks, contributors, commits, issues, and pull requests on GitHub. Our finding is not exactly in line with the prior work by Cruz et al. [55], in which they only found such correlation with contributors and commits but not other metrics. We believe this inconsistency is caused by similar reasons discussed in Section 7.4.1.

We presented this correlation to the survey participants and asked for their opinions as to the possible explanations. 57% (84/148) of the respondents believe that there is a cause-and-effect relationship between test automation and popularity. The causation, however, could be direct, reverse, bidirectional, etc., as explained by some of the respondents below:

"I would say they have a direct connection since the quality and rigidness of the app's code can definitely influence an app's popularity." (direct)

"First you build the app, then it gets popular, then you get resources/motivation to increase it's quality. That's when you go to UI tests." (reverse)

"Projects that become popular end up writing more tests because they need to ensure the stability of the project. As the project becomes more stable (due to more testing) it provides a positive feedback loop. The project, in part, is more likely to be popular if it is perceived as stable, and testing helps to increase that stability." (bidirectional)

On the other hand, 34% (50/148) of the respondents consider this correlation to be more of a connection than causation. For example, the following responses claim common causes for them:

"Common cause: Experienced developer who cares about making code evolvable."

"Popular projects are usually bigger, with multiple developers and with more management. Tests is just a part of that process."

Table 7.13: Impact of Having Tests on the User Satisfaction of Apps. R_w: Apps with Tests. R_o: Apps Without Tests.

	Size	Mean	Median	p-value
R_w	211	4.14	4.25	0.0948
R_o	211	4.2	4.32	
Δ		-0.06	-0.07	

Observation 9: Popular projects are more likely to adopt test automation practices. 57% of the developers believe it implies causality between them.

RQ10. How does test automation relate to user satisfaction?

Following the same steps, we conducted statistical analysis to investigate whether test automation relates to user satisfaction in terms of Google Play ratings. As shown in Table 7.13, we do not find the association between them with statistical significance.

Surprisingly, when we presented this to the survey participants and asked for their opinions, 52% (77/148) of the respondents believe that test automation and user ratings should be somehow related. Namely, the developers do not believe our finding is correct. Examples of their reasons are as follows:

"I think it would depend on the type of application. Games and such are harder to test and the quality of test does not correlate with how fun the game is. For a banking application tests are essential and do effect the quality of the final product."

"Play Store ratings are a noisy metric of app quality and overall user experience, so the no apparent correlation doesn't convince me that app quality isn't impacted at least somewhat by automated testing"

Observation 10: Users' satisfaction with apps appears to be unrelated to the adoption of automated testing practices in their development, while half of the developers think differently.

7.5 Threats to Validity

External validity. The major external validity is the generalization of our findings to all open-source Android apps. We mitigated this threat by including more than $12,000$ apps that vary in terms of size, created year, category, published market, and popularity metrics on GitHub. However, findings in this study may not be applicable to trivial apps or commercial apps developed privately. Furthermore, the respondents of our survey may not be representative of the entire developer community of the subject apps, or the global community of Android app developers. We tried to reduce this threat by collecting the responses of 148 developers from 45 countries with various years of professional experience. The number of responses to our survey is also comparable to other similar studies of mobile developers [77, 90, 81].

Internal validity. We proposed certain heuristics to automatically identify non-trivial apps. While we may have missed some complex and published apps, e.g., apps with single Activity and multiple fragments, we believe that the findings in this chapter are still useful for practitioners and researchers regarding test automation. Moreover, we automatically determine the number of test cases contained in a repository based on the assumption that the test cases are written in JUnit-based testing frameworks. While JUnit-based testing frameworks overwhelmingly dominate Android app testing (e.g., 97% to 100% according to a prior study [55]), it is possible that some test cases built on top of other types of frameworks are not included in our study. To mitigate this threat, we manually verified a small set of projects in our dataset and did not find any missed test cases. As a result, we argue that such cases are rare and would not significantly impact our conclusions.

7.6 Conclusion

This chapter provides a holistic view regarding how and why test automation is practically adopted in open-source Android apps. With the analysis of more than $12,000$ non-trivial apps on GitHub and a survey of 148 developers of these apps, we investigated (1) the trends in the adoption of test automation; (2) working habits of mobile app developers; and (3) the correlation between the adoption of test automation and the popularity of projects. Among others, we found that: (1) only 8% of the non-trivial apps contain automated tests; (2) developers tend to follow the same test automation practices across apps; and (3) popular projects are more likely to adopt test automation practices. We believe the findings in this chapter shed light on the current practices and future research directions pertaining to test automation for mobile app development. In our future work, we plan to incorporate additional open-source projects, such as those hosted on Bitbucket, and investigate new research questions, e.g., questions related to the interplay between test automation techniques and continuous integration practices.

Chapter 8

Conclusion

Reusability is the core concern for both practitioners and researchers in software engineering. Case in point, code reuse is a standard software engineering practice nowadays, through which quality software is built on reliable and practically tested code snippets, libraries, or frameworks. Interestingly, software testing is still conducted in a relatively primitive way. In other words, developers either test their software manually or write the scripted/automated tests from scratch. In fact, just like code reuse that prevents useless effort in reinventing the wheel, test reuse has great potential to be a common practice to reduce the manual effort of software testing, the most expensive activity in the software development life cycle.

In this dissertation, I analyzed the limitations of current automated input generation techniques for software testing, and proposed to address these limitations by test reuse. I presented three concrete scenarios that automated UI testing can benefit from test reuse: (1) test transfer across similar Android apps; (2) test transfer for an app from web to Android; and (3) feature-based test augmentation for Android apps. To demonstrate the feasibility of these scenarios, three automated tools, namely CraftDroid, TransDroid, and Route, were developed and evaluated with real-world applications and test cases. Furthermore, I conducted a large-scale empirical study focusing on non-trivial and open-source Android apps to investigate how and why test automation is practically adopted. Together with the proposed techniques, this empirical study shed light on future research directions pertaining to automated software testing. In the following sections, I conclude my dissertation by enumerating the contributions of my work and avenues for future work.

8.1 Research Contributions

- *Fundamental contribution to UI test reuse.* I designed novel algorithms for the three proposed scenarios of UI test reuse. Although the algorithms were evaluated on specific platforms such as Android, they addressed key challenges shared by all types of UI test reuse. The challenges include: (1) how to map GUI controls between similar apps; (2) how to deal with incompatible actions between the source and the target app; and (3) how to properly reuse existing oracles. As a result, the proposed algorithms are theoretically applicable to any GUI-based software.

- *Tools and experiments.* The proposed algorithms were implemented and empirically evaluated on hundreds of test cases with appropriate metrics to demonstrate their effectiveness and efficiency. Furthermore, the developed tools are publicly released [4, 14, 19] to help other researchers reuse and extend the proposed approaches and build more advanced techniques on top of them.

- *Large-scale empirical study focusing on non-trivial and real-world subjects.* I reported on the first large-scale analysis in over 12,000 open-source projects from 16 app markets and spanning a period of 5 years. The findings in this study are helpful for mobile test reuse and a number of research topics such as automated program repair, mutation testing, and regression test management.

- *Dataset*

- A dataset of 50 UI tests for Android Apps in five categories, annotated with the mapping of the included GUI events for each pair of the tests under the same category [4]. This dataset can be reused to evaluate other intra-platform test transfer techniques.
- A dataset of 220 UI tests for ten pairs of web and Android app [14] that can be reused to evaluate other inter-platform (i.e., web-to-Android) test transfer techniques.
- A dataset of 44 original and 132 augmented UI tests for 6 apps [19] that can be reused to evaluate other feature-based test augmentation techniques.
- A dataset of 12,562 Android projects from GitHub regarding their adoption of unit and UI automated testing [30], as well as the survey response of 148 developers of these projects.

8.2 Future Work

Identification and Prioritization of Reusable Tests. Test reuse or transfer is only meaningful when applied to apps sharing similar features. In this dissertation, I relied on manual inspection and the app store's default categories to determine suitable apps and their tests for reuse. To make test transfer a practical solution, we need an approach or a ranking algorithm to automatically identify and prioritize reusable tests from other apps for the app under test (AUT). Since software contains plenty of textual information by its nature (e.g., in the source code or documentation), natural language processing techniques, particularly unsupervised document classification or text clustering, can be leveraged to develop the proposed algorithm. The reason for adopting unsupervised approaches is that we do not need static labels or data annotation for the potentially reusable tests, but merely how possible the tests can be transferred to the AUT. This possibility will be primarily determined by the similarity between the AUT and the functionality (including the source code and the rendered GUI screens) executed by the tests. To that end, the algorithm will first perform both static and dynamic program analyses to retrieve textual information from the AUT and the potentially reusable tests, by considering their related artifacts, e.g., source code, documentation, and user reviews. Next, several well-adopted techniques in unsupervised text classification, including statistical (e.g., [120, 56]), probabilistic (e.g., [45]), and neural network-based (e.g., [101, 84]) approaches, will be considered to convert the collected text into real number vectors in a vector space. The similarity between the AUT and the candidate tests can be effectively calculated as the proximity of their corresponding vectors.

Reusing Other Associated Artifacts. Apart from existing test cases, other associated artifacts such as bug reports and video recording of software usage are potentially reusable sources for generating meaningful and usage-based tests. Bug reports, while rarely considered, provide invaluable hints for test generation. First, they are submitted by users and usually reflect the use-case scenarios of the application. Secondly, certain types of bugs are common and can be observed in many different applications [130, 33]. For example, a recent study [124] shows that bug reports of a mobile app are very helpful for developers to find similar bugs when they test other similar apps. Finally, many software projects may not have automated tests but a considerable amount of bug reports. While there are techniques (e.g., [134]) to generate executable tests from bug reports and try to reproduce the bug on *the same app*, reusing bug reports to create tests for *other similar apps* has not been explored. Such reuse is feasible if we can leverage the techniques in my previous work to map the GUI controls mentioned in bug reports to the GUI controls rendered by the application under test. In addition to bug reports, screen recordings of application usage are common artifacts in crowdsourcing testing and bug reporting platforms [21, 22]. Since how to translate such videos into executable scenarios has been discussed [44], we can further extend the existing techniques to explore the possibility of reusing screen recordings to generate tests for other similar apps.

Bibliography

[1] https://developer.android.com/studio/test/monkey.

[2] https://firebase.google.com/docs/test-lab/android/robo-ux-test.

[3] https://www.alexa.com/topsites/countries/US.

[4] https://github.com/seal-hub/CraftDroid.

[5] https://play.google.com/store/apps/details?id=com.rainbowshops.

[6] https://play.google.com/store/apps/details?id=com.yelp.android.

[7] https://developer.android.com/studio/command-line/adb.html.

[8] https://github.com/appium/appium.

[9] https://en.wikipedia.org/wiki/Pearson_correlation_coefficient.

[10] https://wordpress.org/.

[11] https://en.wikipedia.org/wiki/Wikipedia.

[12] https://twitter.com.

[13] https://zoom.us/.

[14] https://sites.google.com/view/ase21-transdroid.

[15] https://code.google.com/archive/p/word2vec/.

[16] https://www.soapui.org/docs/functional-testing/validating-messages/getting-started-with-assertions.html/#3-Common-Assertions.

[17] https://www.selenium.dev/.

[18] https://chromedriver.chromium.org/home.

[19] https://sites.google.com/view/route.

[20] https://developer.android.com/training/testing/espresso.

[21] `https://bugclipper.com`.

[22] `https://www.testfairy.com`.

[23] Android-x86 - porting android to x86. https://www.android-x86.org/.

[24] https://developer.android.com/guide/components/fundamentals.html.

[25] https://developer.android.com/guide/topics/manifest/manifest-intro.

[26] https://developer.android.com/studio/releases.

[27] https://developer.android.com/studio/run/managing-avds.

[28] https://www.wsj.com/articles/iowa-caucus-results-delayed-by-apparent-app-issue-11580801699.

[29] https://www.wsj.com/articles/testing-could-have-prevented-iowa-caucus-app-failure-experts-say-11580856659.

[30] https://github.com/seal-hub/ase20empirical, 2020.

[31] `https://play.google.com/store/apps/details?id=daldev.android.gradehelper`, 2021.

[32] 360logica. A sneak peek into test framework and testing pyramid. https://www.360logica.com/blog/sneak-peek-test-framework-test-pyramid-testing-pyramid/, 2020.

[33] C. Q. Adamsen, G. Mezzetti, and A. Møller. Systematic execution of android test suites in adverse conditions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 83–93, New York, NY, USA, 2015. ACM.

[34] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 468–471, New York, NY, USA, 2016. ACM.

[35] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon. Mobiguitar: Automated model-based testing of mobile apps. *IEEE Software*, 32(5):53–59, Sept 2015.

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

[37] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 59:1–59:11, New York, NY, USA, 2012. ACM.

[38] AndroZoo. Androzoo markets. https://androzoo.uni.lu/markets, 2020.

[39] B. Baudry, F. Fleurey, J.-M. Jézéquel, and Y. Le Traon. From genetic to bacteriological algorithms for mutation-based testing. *Software Testing, Verification and Reliability*, 15(2):73–96, 2005.

[40] F. Behrang and A. Orso. Automated test migration for mobile apps. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, pages 384–385, New York, NY, USA, 2018. ACM.

[41] F. Behrang and A. Orso. Test migration for efficient large-scale assessment of mobile app coding assignments. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 164–175, New York, NY, USA, 2018. ACM.

[42] F. Behrang and A. Orso. Test migration between mobile apps with similar functionality. In *Proceedings of the The 34th IEEE/ACM International Conference on Automated Software Engineering*, ASE '19, 2019. To appear.

[43] M. Beller, G. Gousios, A. Panichella, and A. Zaidman. When, how, and why developers (do not) test in their ides. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 179–190, New York, NY, USA, 2015. Association for Computing Machinery.

[44] C. Bernal-Cárdenas, N. Cooper, K. Moran, O. Chaparro, A. Marcus, and D. Poshyvanyk. Translating video recordings of mobile app usages into replayable scenarios. In *Proceedings of the 42nd International Conference on Software Engineering*, ICSE '20, 2020.

[45] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent Dirichlet Allocation. *J. Mach. Learn. Res.*, 3:993–1022, Mar. 2003.

[46] R. Bloem, R. Koenighofer, F. Röck, and M. Tautschnig. Automating test-suite augmentation. In *2014 14th International Conference on Quality Software*, pages 67–72, 2014.

[47] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè. Automatic workarounds: Exploiting the intrinsic redundancy of web applications. *ACM Trans. Softw. Eng. Methodol.*, 24(3), May 2015.

[48] W. Choi, G. Necula, and K. Sen. Guided gui testing of android apps with minimal restart and approximate learning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 623–640, New York, NY, USA, 2013. ACM.

[49] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for android: Are we there yet? (e). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE '15, pages 429–440, Washington, DC, USA, 2015. IEEE Computer Society.

[50] S. R. Choudhary, M. R. Prasad, and A. Orso. Crosscheck: Combining crawling and differencing to better detect cross-browser incompatibilities in web applications. In *Verification and Validation 2012 IEEE Fifth International Conference on Software Testing*, page 171–180, Apr 2012.

[51] S. R. Choudhary, M. R. Prasad, and A. Orso. X-pert: Accurate identification of cross-browser issues in web applications. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 702–711, 2013.

[52] S. R. Choudhary, H. Versee, and A. Orso. Webdiff: Automated identification of cross-browser issues in web applications. In *2010 IEEE International Conference on Software Maintenance (ICSM)*, page 1–10, Sep 2010.

[53] M. Cohn. *Succeeding with agile: software development using Scrum.* Pearson Education, 2010.

[54] R. Coppola, M. Morisio, M. Torchiano, and L. Ardito. Scripted gui testing of android open-source apps: evolution of test code and fragility causes. *Empirical Software Engineering*, 24(5):3205–3248, Oct 2019.

[55] L. Cruz, R. Abreu, and D. Lo. To the attention of mobile software developers: guess what, test your app! *Empirical Software Engineering*, 24(4):2438–2468, Aug 2019.

[56] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American society for information science*, 41(6):391–407, 1990.

[57] Z. Dong, M. Böhme, L. Cojocaru, and A. Roychoudhury. Time-travel testing of android apps. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 481–492, New York, NY, USA, 2020. Association for Computing Machinery.

[58] P. M. Duvall, S. Matyas, and A. Glover. *Continuous integration: improving software quality and reducing risk.* Pearson Education, 2007.

[59] M. Ermuth and M. Pradel. Monkey see, monkey do: Effective generation of gui tests with inferred macro events. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 82–93, New York, NY, USA, 2016. ACM.

[60] C. Escobar-Velásquez, M. Osorio-Riaño, and M. Linares-Vásquez. Mutapk: Source-codeless mutant generation for android apps. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1090–1093, 2019.

[61] F-Droid. F-droid - free and open source android app repository. https://f-droid.org, 2020.

[62] M. Fazzini and A. Orso. Automated cross-platform inconsistency detection for mobile apps. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, page 308–318. IEEE Press, Oct 2017.

[63] B. Fling. A brief history of mobile. https://www.oreilly.com/library/view/mobile-design-and/9780596806231/ch01.html, 2009.

[64] M. Fowler. Test pyramid. https://martinfowler.com/bliki/testpyramid.html, 2020.

[65] G. Fraser and A. Zeller. Generating parameterized unit tests. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, page 364–374, New York, NY, USA, 2011. Association for Computing Machinery.

[66] GHTorrent. Ghtorrent downloads. https://ghtorrent.org/downloads.html, 2020.

[67] Google. Advanced android in kotlin: Testing basics. https://codelabs.developers.google.com/codelabs/advanced-android-kotlin-training-testing-basics/index.html#4, 2020.

[68] Google. Fundamentals of testing. https://developer.android.com/training/testing/fundamentals, 2020.

[69] G. Gousios. The ghtorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 233–236, Piscataway, NJ, USA, 2013. IEEE Press.

[70] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan. Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, pages 204–217, New York, NY, USA, 2014. ACM.

[71] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, page 60–71, USA, 2003. IEEE Computer Society.

[72] D. Hillis. Mobile internet era: Planning your mobile strategy. https://www.cmswire.com/cms/web-engagement/mobile-internet-era-planning-your-mobile-strategy-010147.php, 2011.

[73] G. Hu, L. Zhu, and J. Yang. Appflow: Using machine learning to synthesize robust, reusable ui tests. In *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, ESEC/FSE 2018, 2018.

[74] R. Jabbarvand and S. Malek. μdroid: an energy-aware mutation testing framework for android. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 208–219. ACM, 2017.

[75] C. S. Jensen, M. R. Prasad, and A. Møller. Automated testing with targeted event sequence generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 67–77, New York, NY, USA, 2013. ACM.

[76] M. E. Joorabchi, M. Ali, and A. Mesbah. Detecting inconsistencies in multi-platform mobile apps. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, page 450–460, Nov 2015.

[77] M. E. Joorabchi, A. Mesbah, and P. Kruchten. Real challenges in mobile app development. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, Maryland, USA, October 10-11, 2013*, pages 15–24, 2013.

[78] JUnit. Junit. https://junit.or, 2020.

[79] K. Kapelonis. Software testing anti-patterns. http://blog.codepipes.com/testing/software-testing-antipatterns.html, 2020.

[80] Y. Kim, Z. Zu, M. Kim, M. B. Cohen, and G. Rothermel. Hybrid directed test suite augmentation: An interleaving framework. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 263–272, 2014.

[81] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo. Understanding the test automation culture of app developers. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, pages 1–10. IEEE, 2015.

[82] Y. Koroglu, A. Sen, O. Muslu, Y. Mete, C. Ulker, T. Tanriverdi, and Y. Donmez. Qbe: Qlearning-based exploration of android applications. In *Software Testing, Verification and Validation (ICST), 2018 IEEE 11th International Conference on*, pages 105–115. IEEE, 2018.

[83] E. Kreyszig. *Advanced Engineering Mathematics, 10th Eddition.* Wiley, 2009.

[84] M. J. Kusner, Y. Sun, N. I. Kolkin, and K. Q. Weinberger. From Word Embeddings to Document Distances. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pages 957–966. JMLR.org, 2015.

[85] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.

[86] J. Lin, R. Jabbarvand, and S. Malek. Test transfer across mobile apps through semantic mapping. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 42–53, 2019.

[87] J.-W. Lin, N. Salehnamadi, and S. Malek. Test Automation in Open-Source Android Apps: A Large-Scale Empirical Study. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020.

[88] 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 *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 233–244, New York, NY, USA, 2017. Association for Computing Machinery.

[89] M. Linares-Vásquez, M. White, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk. Mining android app usages for generating actionable gui-based execution scenarios. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 111–122, Piscataway, NJ, USA, 2015. IEEE Press.

[90] M. Linares-Vásquez, C. Bernal-Cardenas, K. Moran, and D. Poshyvanyk. How do developers test android applications? In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 613–622, Sept 2017.

[91] M. Linares-Vásquez, K. Moran, and D. Poshyvanyk. Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 399–410, Sep. 2017.

[92] P. Liu, X. Zhang, M. Pistoia, Y. Zheng, M. Marques, and L. Zeng. Automatic text input generation for mobile testing. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 643–653, Piscataway, NJ, USA, 2017. IEEE Press.

[93] C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek. DéJàVu: A Map of Code Duplicates on GitHub. *Proc. ACM Program. Lang.*, 1(OOPSLA):84:1–84:28, Oct. 2017.

[94] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 224–234, New York, NY, USA, 2013. ACM.

[95] R. Mahmood, N. Mirzaei, and S. Malek. Evodroid: Segmented evolutionary testing of android apps. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 599–609, New York, NY, USA, 2014. ACM.

[96] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *Ann. Math. Statist.*, 18(1):50–60, 03 1947.

[97] K. Mao, M. Harman, and Y. Jia. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 94–105, New York, NY, USA, 2016. ACM.

[98] L. Mariani, M. Pezzè, and D. Zuddas. Augusto: Exploiting popular functionalities for the generation of semantic gui tests with oracles. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 280–290, New York, NY, USA, 2018. ACM.

[99] A. M. Memon and M. B. Cohen. Automated testing of gui applications: Models, tools, and controlling flakiness. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, page 1479–1480. IEEE Press, 2013.

[100] A. Mesbah and M. R. Prasad. Automated cross-browser compatibility testing. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, page 561–570. Association for Computing Machinery, May 2011.

[101] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 3111–3119. Curran Associates, Inc., 2013.

[102] A. Milani Fard, M. Mirzaaghaei, and A. Mesbah. Leveraging existing tests in automated test generation for web applications. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE'14, page 67–78. ACM, 2014.

[103] N. Mirzaei, H. Bagheri, R. Mahmood, and S. Malek. Sig-droid: Automated system input generation for android applications. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 461–471, Nov 2015.

[104] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek. Reducing combinatorics in gui testing of android applications. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 559–570. IEEE, 2016.

[105] Mockito. Most popular mocking framework for java. https://github.com/mockito/mockito, 2020.

[106] R. M. Moreira, A. C. Paiva, and A. Memon. A pattern-based approach for gui modeling and testing. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, pages 288–297. IEEE, 2013.

[107] D. Nisbet. Test automation basics – levels, pyramids & quadrants. http://www.duncannisbet.co.uk/test-automation-basics-levels-pyramids-quadrants, 2020.

[108] C. Onyaem. Separate unit, integration, and functional tests for continuous delivery. https://medium.com/pacroy/separate-unit-integration-and-functional-tests-for-continuous-delivery-f4dc240d8f2f, 2020.

[109] Oracle. Oracle vm virtualbox. https://www.virtualbox.org/, 2021.

[110] F. Pecorelli, G. Catolino, F. Ferrucci, A. De Lucia, and F. Palomba. Testing of mobile applications in the wild: A large-scale empirical study on android apps. In *28th IEEE/ACM International Conference on Program Comprehension (ICPC)*, 2020.

[111] M. Pezze, K. Rubinov, and J. Wuttke. Generating effective integration test cases from unit ones. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 11–20, Los Alamitos, CA, USA, mar 2013. IEEE Computer Society.

[112] X. Qin, H. Zhong, and X. Wang. Testmig: Migrating gui test cases from ios to android. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, pages 284–295, New York, NY, USA, 2019. ACM.

[113] Qualtrics. Qualtrics survey software. https://www.qualtrics.com/, 2020.

[114] A. Rau, J. Hotzkow, and A. Zeller. Transferring tests across web applications. In T. Mikkonen, R. Klamma, and J. Hernández, editors, *Web Engineering*, pages 50–64, Cham, 2018. Springer International Publishing.

[115] Robolectric. Test-drive your android code. http://robolectric.org/, 2020.

[116] A. Rosenfeld, O. Kardashov, and O. Zang. Automation of android applications functional testing using machine learning activities classification. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, MOBILESoft '18, pages 122–132, New York, NY, USA, 2018. ACM.

[117] S. Roy Choudhary, M. R. Prasad, and A. Orso. Cross-platform feature matching for web applications. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, page 82–92. Association for Computing Machinery, Jul 2014.

[118] S. Rusticus and C. Lovato. Impact of sample size and variability on the power and type i error rates of equivalence tests: A simulation study. *Practical Assessment, Research and Evaluation*, 19, 01 2014.

[119] N. Salehnamadi, A. Alshayban, J.-W. Lin, I. Ahmed, S. Branham, and S. Malek. Latte: Use-case and assistive-service driven automated accessibility testing framework for android. In *The 2021 ACM Conference on Human Factors in Computing Systems*, CHI '21. Association for Computing Machinery, 2021.

[120] G. Salton and M. J. McGill. Introduction to Modern Information Retrieval. 1986.

[121] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 218–227, 2008.

[122] S. S. SHAPIRO and M. B. WILK. An analysis of variance test for normality (complete samples)†. *Biometrika*, 52(3-4):591–611, 12 1965.

[123] 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*, ESEC/FSE 2017, pages 245–256, New York, NY, USA, 2017. ACM.

[124] S. H. Tan and Z. Li. Collaborative bug finding for android apps. In *Proceedings of the 42nd International Conference on Software Engineering*, ICSE '20, 2020.

[125] N. Tillmann and W. Schulte. Unit tests reloaded: parameterized unit testing with symbolic execution. *IEEE Software*, 23(4):38–47, 2006.

[126] P. Tonella, R. Tiella, and C. D. Nguyen. Interpolated n-grams for model based testing. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 562–572, New York, NY, USA, 2014. ACM.

[127] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press, 1999.

[128] W. Yang, M. R. Prasad, and T. Xie. A grey-box approach for automated gui-model generation of mobile applications. In V. Cortellessa and D. Varró, editors, *Fundamental Approaches to Software Engineering*, pages 250–265, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[129] S. Yoo and M. Harman. Test data regeneration: Generating new test data from existing test data. *Softw. Test. Verif. Reliab.*, 22(3):171–201, May 2012.

[130] R. N. Zaeem, M. R. Prasad, and S. Khurshid. Automated generation of oracles for testing user-interaction features of mobile apps. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, ICST '14, pages 183–192, Washington, DC, USA, 2014. IEEE Computer Society.

[131] H. Zhang and A. Rountev. Analysis and testing of notifications in android wear applications. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 347–357, Piscataway, NJ, USA, 2017. IEEE Press.

[132] P. Zhang and S. Elbaum. Amplifying tests to validate exception handling code. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 595–605, 2012.

[133] Y. Zhao, J. Chen, A. Sejfia, M. S. Laser, J. Zhang, F. Sarro, M. Harman, and N. Medvidovic. Fruiter–a framework for evaluating ui test reuse. In *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, ESEC/FSE 2020, 2020.

[134] Y. Zhao, T. Yu, T. Su, Y. Liu, W. Zheng, J. Zhang, and W. G.J. Halfond. Recdroid: Automatically reproducing android application crashes from bug reports. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 128–139, 2019.