ONLINE CONSTRUCTION OF ANDROID APPLICATION TEST SUITES

David T. Adamo, Jr.

Dissertation Prepared for the Degree of

DOCTOR OF PHILOSOPHY

UNIVERSITY OF NORTH TEXAS

December 2017

APPROVED:

Renée Bryce, Co-Advisor
Barrett Bryant, Co-Advisor
Philip Sweany, Committee Member
Hyunsook Do, Committee Member
Barrett Bryant, Chair of the Department of Computer Science and Engineering
Costas Tsatsoulis, Dean of the College of Engineering
Victor Prybutok, Dean of the Toulouse Graduate School

Adamo Jr., David T. *Online Construction of Android Application Test Suites*. Doctor of Philosophy (Computer Science and Engineering), December 2017, 102 pp., 29 tables, 34 figures, 78 numbered references.

Mobile applications play an important role in the dissemination of computing and information resources. They are often used in domains such as mobile banking, e-commerce, and health monitoring. Cost-effective testing techniques in these domains are critical. This dissertation contributes novel techniques for automatic construction of mobile application test suites. In particular, this work provides solutions that focus on the prohibitively large number of possible event sequences that must be sampled in GUI-based mobile applications. This work makes three major contributions: (1) an automated GUI testing tool, Autodroid, that implements a novel online approach to automatic construction of Android application test suites (2) probabilistic and combinatorial-based algorithms that systematically sample the input space of Android applications to generate test suites with GUI/context events and (3) empirical studies to evaluate the cost-effectiveness of our techniques on real-world Android applications. Our experiments show that our techniques achieve better code coverage and event coverage compared to random test generation. We demonstrate that our techniques are useful for automatic construction of Android application test suites in the absence of source code and preexisting abstract models of an Application Under Test (AUT). The insights derived from our empirical studies provide guidance to researchers and practitioners involved in the development of automated GUI testing tools for Android applications.

Copyright 2017

by

David T. Adamo Jr.

ACKNOWLEDGMENTS

Thank you to my advisors Renée Bryce and Barrett Bryant for giving me the support and encouragement I needed throughout the PhD program.

Thank you to the members of my PhD committee, Hyunsook Do and Philip Sweany, for providing valuable feedback on this PhD dissertation.

Thank you to the members of the Research Innovations in Software Engineering (RISE) lab, especially Dmitry Nurmuradov, Quentin Mayo, Sreedevi Koppula and Shraddha Piparia, for contributing to an enjoyable and inspiring work environment.

Thank you to Ultimate Software Group Inc. for providing a work environment that encourages professional and academic growth.

Finally, thank you to my family for teaching me the value of education and for always being supportive.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS iii

LIST OF TABLES viii

LIST OF FIGURES x

CHAPTER 1 INTRODUCTION 1

1.1. Motivation 1

1.2. The Automated Test Generation Problem 2

1.3. Contributions 4

1.3.1. Online Construction of Reusable Test Suites for Android Apps 4

1.3.2. Test Suite Construction Algorithms and Heuristics 4

1.3.3. Autodroid: An Automated GUI Testing Tool for Android Apps 4

1.3.4. Empirical Studies 5

1.4. Organization 5

CHAPTER 2 BACKGROUND AND RELATED WORK 6

2.1. Automated GUI Testing Techniques 6

2.1.1. Offline (Model-based) Testing Techniques 6

2.1.2. Online (Dynamic) Testing Techniques 6

2.2. Random Testing 7

2.3. Combinatorial Interaction Testing 8

2.4. Related Work in Automated GUI Testing 10

2.5. Android Mobile Applications 12

2.6. GUI Event Interaction in Android Applications 13

2.7. Context-Sensitivity in Android Applications 15

2.8. Automated GUI Testing of Android Applications 18

CHAPTER 3 ONLINE CONSTRUCTION OF REUSABLE TEST SUITES FOR

ANDROID APPLICATIONS 20
3.1. An Event Sequence Metamodel for Online GUI Testing 21
3.2. Online Event Sequence Generation 25
3.3. Random-based Test Suite Construction 26
3.4. Tool Implementation 28
3.5. Experiments 29
3.5.1. Subject Applications 30
3.5.2. Experimental Setup 30
3.5.3. Results 31
3.5.4. Discussion and Implications 32
3.5.5. Threats to Validity 33
3.6. Summary and Conclusions 33

CHAPTER 4 FREQUENCY-BASED TEST SUITE CONSTRUCTION 34
4.1. Test Suite Construction Algorithm 35
4.1.1. Frequency Weighted Event Selection 36
4.1.2. Minimum Frequency Event Selection 38
4.2. Implementation 39
4.3. Experiments 39
4.3.1. Research Questions 39
4.3.2. Subject Applications 39
4.3.3. Experimental Setup 40
4.3.4. Variables and Measures 40
4.3.5. Data collection 42
4.3.6. Statistical tests 42
4.3.7. Results 42
4.3.8. Discussion and Implications 51
4.3.9. Threats to Validity 52
4.4. Summary and Conclusions 53

CHAPTER 5 COMBINATORIAL-BASED TEST SUITE CONSTRUCTION 54
5.1. Combinatorial-based Test Suite Construction Algorithm 55
5.1.1. Candidate Event Selection 59
5.2. Experiments 59
5.2.1. Research Questions 59
5.2.2. Subject Applications 60
5.2.3. Experimental Setup 60
5.2.4. Variables and measures 61
5.2.5. Implementation 62
5.2.6. Data collection 62
5.2.7. Statistical tests 62
5.2.8. Results 63
5.2.9. Discussion and Implications 72
5.2.10. Threats to Validity 74
5.3. Summary and Conclusions 74

CHAPTER 6 TESTING CONTEXT-SENSITIVE BEHAVIOR IN ANDROID APPLICATIONS 76
6.1. Context Modelling 77
6.2. Definitions 78
6.3. Test Suite Construction Framework 79
6.3.1. Pairwise Event Selection 81
6.3.2. Test Generation Techniques 82
6.3.3. Framework Implementation 83
6.4. Experiments 84
6.4.1. Research Questions 84
6.4.2. Subject Application 84
6.4.3. Experimental Setup 85
6.4.4. Results and Discussion 86

6.4.5. Threats to Validity 89
6.5. Summary and Conclusions 89
CHAPTER 7 CONCLUSIONS AND FUTURE WORK 90
7.1. Summary of Contributions 90
7.2. Future Work 93
REFERENCES 95

LIST OF TABLES

Page

Table 2.1. Combinatorial testing model with four parameters and three values for each parameter 9

Table 2.2. A 2-way interaction test suite (covering array) 9

Table 2.3. GUI event sequences and interaction-based behavior in a real-world Android app 14

Table 3.1. Example test case with two events 24

Table 3.2. Characteristics of selected Android apps 30

Table 3.3. Mean block coverage for Monkey and Autodroid across 10 test suites for each subject application 31

Table 4.1. Characteristics of selected Android apps 39

Table 4.2. Mean block coverage for *Rand*, *FreqWeighted* and *MinFrequency* test suites 43

Table 4.3. Statistical comparison of block coverage (BC) values for *Rand*, *FreqWeighted* and *MinFrequency* test suites 43

Table 4.4. Mean method coverage for *Rand*, *FreqWeighted* and *MinFrequency* test suites 44

Table 4.5. Statistical comparison of method coverage (MC) values for *Rand*, *FreqWeighted* and *MinFrequency* test suites 45

Table 4.6. Mean APBC values for *Rand*, *FreqWeighted* and *MinFrequency* test suites 46

Table 4.7. Statistical comparison of APBC values for *Rand*, *FreqWeighted* and *MinFrequency* test suites 47

Table 4.8. Average number of distinct events (event coverage) for *Rand*, *FreqWeighted* and *MinFrequency* test suites 50

Table 4.9. Statistical comparison of event coverage (EC) values for *Rand*, *FreqWeighted* and *MinFrequency* test suites 51

Table 5.1. Characteristics of selected Android apps 60

Table 5.2. Mean block coverage of random-based and combinatorial-based test suites 64
Table 5.3. Statistical comparison of block coverage (BC) values for random-based, 2-way combinatorial-based and 3-way combinatorial-based test suites 64
Table 5.4. Mean method coverage of random-based and combinatorial-based test suites 65
Table 5.5. Statistical comparison of method coverage (MC) values for random-based, 2-way combinatorial-based and 3-way combinatorial-based test suites 65
Table 5.6. Mean APBC values for the random-based and combinatorial-based test suites 67
Table 5.7. Statistical comparison of APBC values for random-based, 2-way combinatorial-based and 3-way combinatorial-based test suites 68
Table 5.8. Average number of distinct events covered (rounded to whole numbers) across 10 test suites for each subject application and technique 71
Table 5.9. Statistical comparison of event coverage (EC) for random-based, 2-way combinatorial-based and 3-way combinatorial-based test suites 72
Table 6.1. Combinatorial testing model with four context variables and two values for each variable 77
Table 6.2. A 2-way covering array that defines six contexts 78
Table 6.3. Test generation techniques with corresponding parameter specifications 82
Table 6.4. Summary block coverage statistics across 10 test suites for each technique 86
Table 6.5. Exceptions found by each test generation technique 88

LIST OF FIGURES

		Page
Figure 2.1.	Example of an Android application GUI	12
Figure 2.2.	GUI state transition graph for a mobile application (Tomdroid)	14
Figure 2.3.	Launching an Android application in two different contexts	16
Figure 2.4.	Clicking a list item with airplane mode ON (no internet access)	17
Figure 2.5.	Clicking a list item with airplane mode OFF (internet access)	17
Figure 3.1.	GUI event sequence metamodel	21
Figure 3.2.	Online event sequence construction	25
Figure 3.3.	Input, output and major components of Autodroid	28
Figure 3.4.	Distribution of block coverage values across 10 suites generated with Autodroid and Monkey for each subject application	32
Figure 4.1.	Examples of APBC measures	41
Figure 4.2.	Boxplot of block coverage values across 10 suites for each subject application and event selection strategy	43
Figure 4.3.	Distribution of method coverage values across 10 suites for each subject application and event selection strategy	45
Figure 4.4.	Boxplot of APBC values across 10 test suites for each app and event selection strategy	46
Figure 4.5.	Coverage-time graph for *Tomdroid*	48
Figure 4.6.	Coverage-time graph for *Loaned*	48
Figure 4.7.	Coverage-time graph for *Budget*	48
Figure 4.8.	Coverage-time graph for *ATimeTracker*	48
Figure 4.9.	Coverage-time graph for *Repay*	49
Figure 4.10.	Coverage-time graph for *Moneybalance*	49
Figure 4.11.	Coverage-time graph for *WhoHasMyStuff*	49
Figure 4.12.	Distribution of event coverage values across 10 suites for each subject application and event selection strategy	50

Figure 5.1. Boxplot of block coverage values across 10 suites for each app and technique 64
Figure 5.2. Boxplot of method coverage values across 11 ten suites for each app and technique 66
Figure 5.3. Boxplot of APBC values across 10 test suites for each app and technique 67
Figure 5.4. Coverage-time graph for *Tomdroid* 69
Figure 5.5. Coverage-time graph for *Loaned* 69
Figure 5.6. Coverage-time graph for *Budget* 69
Figure 5.7. Coverage-time graph for *ATimeTracker* 69
Figure 5.8. Coverage-time graph for *Repay* 70
Figure 5.9. Coverage-time graph for *Moneybalance* 70
Figure 5.10. Coverage-time graph for *WhoHasMyStuff* 70
Figure 5.11. Boxplot of number of distinct events across 10 test suites for each app and technique 71
Figure 6.1. Framework implementation 83
Figure 6.2. Average number of events executed in each context 86

CHAPTER 1

INTRODUCTION

1.1. Motivation

Mobile devices are increasingly powerful tools that provide portable access to computing resources and services. Smart mobile devices provide Operating Systems (OS) that serve as a platform for mobile applications (apps). These mobile applications provide services in critical domains such as ecommerce, mobile banking and mobile health monitoring where a faulty mobile app could lead to devastating consequences for end users and developers. The mobile app market is a $77 billion industry [**27**] and only about 16% of users are likely to try a failing app more than twice [**66**]. Therefore, the success of a mobile app may depend on how thoroughly it is tested.

This dissertation focuses on the Android platform since it currently dominates the mobile Operating System (OS) market worldwide [**33**]. The availability of extensive documentation, development frameworks and app stores has enabled a large number of developers with limited resources to easily build and distribute Android apps. Many developers lack extensive training in software testing and may not have the resources necessary to adequately test their applications before they are released to end users. An analysis of over 600 open source Android apps shows that only about 14% of the apps contain test cases and the majority of the apps with test cases provide less than 40% code coverage [**35**]. These findings suggest that the majority of Android apps are poorly tested. Many mobile app developers choose to manually write test scripts with libraries such as Robotium [**62**], Espresso [**28**] and JUnit [**34**]. An alternative is to use capture-replay tools (such as Robotium Recorder [**61**]) that enable developers to manually execute and record sequences of GUI events for replay at a later time. The significant amount of manual effort that these techniques require often limits developers' ability to author tests that adequately explore the vast input space of a mobile app. Automated GUI testing tools and techniques may minimize the manual effort required to construct effective test cases for mobile apps. This dissertation describes

Autodroid, an automated GUI testing tool for Android apps and investigates several algorithms that systematically explore the input space of GUI-based Android applications to automatically construct cost-effective test suites.

1.2. The Automated Test Generation Problem

This dissertation addresses several challenges that hinder the use of automated test generation techniques within the context of mobile applications. These challenges include the vast input space of GUI-based applications, inadequate tool support for generating reusable test suites, context-sensitivity in mobile applications and limited empirical studies on the effectiveness of various algorithms.

Vast input space. Mobile applications are Event Driven Systems (EDSs) that take event sequences as input and respond by changing their state. Mobile applications typically have a Graphical User Interface (GUI) that enables user interaction via sequences of user actions such as clicking a button or typing text into an input field. Mobile applications conform to platform-specific GUI design patterns that must be considered when developing automated test generation techniques. GUI-based applications are particularly difficult to test because of the prohibitively large number of possible event sequences that make up the input space and must be sampled during test generation. Each possible event sequence in the input space represents a potential test case that may trigger faults in the Application Under Test (AUT). The automatic test generation problem requires solutions that enable efficient exploration of this vast input space, especially when there is insufficient information about the inner workings of the AUT.

Reusable test suites. Any investigation into the cost-effectiveness of various test generation techniques requires adequate tool support. The majority of existing automated GUI testing tools for Android applications (e.g. [**29**, **32**, **42**]) do not generate reexecutable test cases that enable automated regression testing and easy reproduction of failures [**19**]. Testers often perform regression testing by reusing test cases from previous versions of the AUT. Regression testing assesses the continued quality of an AUT after one or more changes in its functionality. In many cases, existing tools generate and execute a single event sequence of

predetermined length rather than a test suite with distinct event sequences (test cases). A single event sequence that contains a large number of events may be difficult to examine and reexecute to reproduce failures. Automated GUI testing tools that generate reusable test cases facilitate automated regression testing, alleviate the difficulty of failure reproduction and provide useful insight into the structure of test cases produced by various test generation techniques.

Context-sensitivity. Mobile applications further complicate the testing process with their ability to respond not just to GUI events, but also context events (e.g. changes in network connectivity, battery levels, location, etc.) to provide context-sensitive functionality to users. Context events often modify one or more context variables (e.g. screen orientation, connectivity status, etc.) that define the *operating context* of a mobile application and may affect its behavior. Faults may occur only in specific operating contexts or as a result of interactions between context variables. The majority of existing research focuses predominantly on GUI events with limited or no consideration for context events and how they affect the behavior of an AUT. Automated testing techniques that integrate context changes into the test generation process may enable cost-effective testing of context-sensitive behavior that may otherwise go untested.

Empirical studies. There is a lack of empirical studies that objectively compare test generation techniques within the context of Android applications. Prior work (e.g. [**19**]) compares different automated GUI testing tools for Android apps. The results of such comparisons are often affected by differences in event abstraction and implementation choices across various tools. While existing empirical studies may provide information regarding what tools work best in particular testing scenarios, they do not provide insight into which algorithms and heuristics are most cost-effective in comparison to random testing. Arcuri et al. [**7**] recommend random test generation as a baseline for empirical evaluation of test generation algorithms especially when such algorithms use randomization. Experiments that minimize the influence of abstraction and implementation differences across tools may enable objective comparisons between various test generation algorithms and heuristics.

This dissertation sets out to investigate the following thesis: **within the context of GUI-based Android applications, probabilistic and combinatorial-based test generation techniques may be used to develop algorithms that significantly outperform random test generation in terms of code coverage and event coverage despite the additional computational overhead required by such techniques**.

1.3. Contributions

1.3.1. Online Construction of Reusable Test Suites for Android Apps

This dissertation describes a novel online approach to automatic construction of reusable Android application test suites. We define an event sequence metamodel that specifies an abstract representation for automatically generated test cases such that each test case that conforms to the metamodel may be automatically reexecuted for automated regression testing or reproduction of failures. We develop a process to automatically construct test suites that conform to our metamodel and demonstrate the feasibility of our approach with real-world Android applications.

1.3.2. Test Suite Construction Algorithms and Heuristics

This dissertation presents new algorithms and heuristics for online construction of Android application test suites. We use probabilistic and combinatorial-based techniques to develop algorithms that automatically construct Android application test suites with distinct event sequences as test cases. We develop a framework that enables automated testing of context-sensitive behavior in Android applications. The framework allows instantiation of multiple online algorithms to generate tests that interleave GUI events and context events in different ways.

1.3.3. Autodroid: An Automated GUI Testing Tool for Android Apps

We implement the algorithms and techniques in this work as part of an automated GUI testing tool, Autodroid, that uses our online approach to automatically generate reusable test suites. Autodroid provides tool support for our empirical studies and enables objective comparisons between several test suite construction algorithms.

1.3.4. Empirical Studies

This dissertation describes empirical studies to evaluate the cost-effectiveness of probabilistic and combinatorial-based algorithms for online construction of Android application test suites. We perform experiments with real-world Android applications and compare our techniques to random test generation in terms of code coverage and event coverage. We also perform experiments to evaluate multiple techniques for online construction of test suites with context events and GUI events. To facilitate objective comparisons between techniques, we implement the techniques in our empirical studies within the same tool and use the same abstractions across all algorithms. Our empirical studies provide insight into which algorithms are most cost-effective for online construction of Android application test suites, given a fixed time budget for testing. The empirical studies in this dissertation provide guidance to researchers and practitioners who are involved in the development of automated GUI testing tools for Android apps.

1.4. Organization

Chapter 2 provides an overview of Android applications and discusses related work in automated GUI testing. Chapter 3 presents our online approach to automatic construction of Android application test suites and describes a tool, Autodroid, that implements our approach. In Chapter 4, we describe and evaluate frequency-based techniques to reduce redundant event selection in random-based test suites and increase code coverage. Chapter 5 describes and evaluates a combinatorial-based technique that maximizes coverage of n-way event combinations and considers the order in which events have previously occured during test suite construction. In Chapter 6, we describe a framework that enables testing of context-sensitive behavior in Android applications. The framework allows testers to instantiate various test generation algorithms that interleave context events and GUI events in different ways. Finally, Chapter 7 concludes the dissertation.

CHAPTER 2

BACKGROUND AND RELATED WORK

This chapter provides an overview of software testing techniques and Android applications. We discuss related work in the area of automated GUI testing and describe GUI testing challenges that motivate the techniques and experiments in this work.

2.1. Automated GUI Testing Techniques

Automated GUI testing techniques generally fall into two categories: offline (model-based) testing techniques and online testing techniques [**3**, **10**]. This section describes both techniques and the practical issues that affect their application to automated GUI testing.

2.1.1. Offline (Model-based) Testing Techniques

Offline testing techniques require a static abstract model of the Application Under Test (AUT) to automatically generate test cases. These techniques often use state machines [**5**, **11**, **70**] or event flow graphs [**48**, **72**, **76**] to generate GUI event sequences. Test generation and test execution are separate activities and test execution occurs after test generation. Abstract models of the AUT may be constructed manually or with *GUI Rippers* that automatically extract models from GUIs. A graph-based model of the AUT enables generation of event sequences that satisfy graph-based coverage criteria such as node coverage or transition coverage. Manual construction of abstract models is time-consuming and automatically constructed models are often inaccurate. Inaccurate models of an AUT's behavior often produce test cases that are only partially executable [**10**]. The majority of existing research in automated GUI testing uses model-based techniques.

2.1.2. Online (Dynamic) Testing Techniques

Online testing techniques do not require a preexisting abstract model of the AUT. During test generation, an online algorithm dynamically analyzes each GUI state of the AUT to identify, select and execute GUI events [**3**, **10**]. Online GUI testing algorithms create event sequence test cases one-event-at-a-time through multiple iterations of event

identification, selection and execution. Event selection in each GUI state may be random or based on some predefined criteria (e.g. prior selection frequency). Test cases generated with online techniques are always fully executable since test generation and execution occur cooperatively based on the actual behavior of the AUT, rather than a possibly inaccurate abstract model.

This dissertation presents new algorithms for online construction of Android application test suites and empirical studies to assess their cost-effectiveness. Our techniques do not require source code analysis or preexisting abstract models of the AUT's behavior. Our algorithms automatically construct test cases one-event-at-a-time with an *event extraction cycle* that iteratively identifies, selects and executes events from the GUI of the application under test.

2.2. Random Testing

Random testing is a simple technique that is often used to automatically test software [**9**, **31**]. Random-based techniques choose tests at random from the input domain of the System Under Test (SUT). Input selection may be based on a defined probability distribution (e.g. uniform, normal, etc.) or an operational profile that represents typical usage of the SUT [**67**]. Miller et al. [**52**] show that random testing finds crashes in MacOS command-line and GUI applications. One criticism of random testing is that it does not leverage any information about the SUT to improve the testing process. Chen et al. [**18**] propose an adaptive approach to random testing that evenly spreads tests across the input domain. Adaptive Random Testing (ART) techniques operate under the assumption that failure-causing inputs tend to cluster within the input domain of an SUT. These techniques often use distance metrics (e.g. euclidean distance) to select test cases that are dissimilar to each other and maximize some notion of test case diversity within a test suite [**41**]. Arcuri et al.[**6**] show that compared to simple random testing, ART techniques are inefficient even for trivial problems because of the repeated distance calculations among test cases. Random testing is often compared to structural testing in terms of code coverage and fault-finding effectivenesss [**9**, **24**]. Naive random testing techniques sometimes perform better than more

sophisticated techniques that incur extra computational overhead [**8**, **63**]. Duran and Ntafos [**24**] show that random testing is a useful testing technique that may detect failures in an SUT with less effort compared to more sophisticated techniques such as partition testing. These observations suggest that any proposed novel technique for test generation should be compared to a random-based technique. Such a comparison is necessary to show that any computational overhead incurred by the novel technique does not outweigh its potential benefits relative to random testing [**7**]. In this work, we introduce Autodroid, an automated testing tool that supports various algorithms for online construction of Android application test suites. We use random-based test suites generated with Autodroid as a baseline to evaluate the test suite construction algorithms in this work.

2.3. Combinatorial Interaction Testing

Combinatorial Interaction Testing (CIT) techniques systematically examine interactions between inputs of a system [**37**, **38**, **59**, **68**, **69**, **74**]. CIT often requires system input to be modeled as parameters and values. Since the number of possible parameter-value combinations may be prohibitively large, CIT techniques use sampling mechanisms that systematically combine parameter-values to form a covering array. A *covering array* $CA(N; t; k; v)$ is an array with N rows and k columns such that each t-tuple occurs at least once within the rows. For a covering array $CA(N; t; k; v)$, k is the number of parameters in the combinatorial input model, v is the number of values associated with each parameter and t is the *strength* of interaction coverage. Each row of the covering array specifies parameter-values for a specific test. The covering array specifies a collection of tests that cover all t-way combinations of parameter values, where t is the number of parameters to combine [**59**]. CIT algorithms aim to cover all t-way parameter-value combinations in as few tests as possible to detect failures triggered by interactions between parameters.

Consider the example of a combinatorial testing model in Table 2.1 with four parameters and three values for each parameter. An example of a pairwise interaction for this input specification is: {*OrderCategory=Rent, Location=London*}. An exhaustive interaction test suite requires 3^4 tests. Table 2.2 shows a 2-way interaction test suite (covering array) that

requires only 9 tests. Any two columns of the array contain all possible value combinations for two specific parameters corresponding to the columns.

Order Category	**Location**	**Order Time**	**Order Type**
Buy	London	Working hours	Online
Sell	New York	Non-working hours	In store
Rent	Seattle	Holiday	Phone

TABLE 2.1. Combinatorial testing model with four parameters and three values for each parameter

Test No.	**Order Category**	**Location**	**Order Time**	**Order Type**
1	Buy	London	Working hours	Online
2	Buy	New York	Non-working hours	In store
3	Buy	Seattle	Holiday	Phone
4	Sell	London	Holiday	In store
5	Sell	New York	Working hours	Phone
6	Sell	Seattle	Non-working hours	Online
7	Rent	London	Non-working hours	Phone
8	Rent	New York	Holiday	Online
9	Rent	Seattle	Working hours	In store

TABLE 2.2. A 2-way interaction test suite (covering array)

Empirical studies show that combinatorial-based testing may be effective for detecting software faults. Kuhn et al. [**37**, **38**] show that faults are often triggered by interactions among six or fewer parameters. There are several algorithms and techniques for automatic generation of covering arrays from combinatorial testing models [**13**–**16**,**20**–**22**,**38**,**39**]. Many of these algorithms focus on testing interactions in systems where inputs are not sequence-based and the order of inputs is not important. Kuhn et al. [**36**] apply combinatorial methods to event sequence testing where the order of events is important. They construct Sequence Covering Arrays (SCAs) that test any t events in every possible t-way order. Their technique is limited to situations where events are not repeated, sequences are of fixed length and there are no constraints on the order in which events occur in a sequence. We extend their technique to the mobile application domain where there are constraints on the order of

events, sequences can have varying lengths and events may be repeated. In Chapters 5 and 6, we describe and evaluate online test suite construction algorithms that use covering arrays and combinatorial methods to automatically construct Android application test suites.

2.4. Related Work in Automated GUI Testing

GUI-based applications are event-driven. Examples of GUI-based applications include desktop applications, web applications and mobile applications. The large number of possible event sequences in a GUI-based application makes testing particularly challenging. For a given AUT, each possible event sequence represents a potential test case. Automated testing techniques may be used to sample the event sequence space of an AUT and select test cases that expose faulty behavior [**12**].

The majority of existing work in automated GUI testing uses model-based testing techniques to automatically generate event sequences. Model-based techniques require construction of static abstract models of the AUT prior to test generation. Prior work describes several types of graph-based models for automated GUI testing. For such graph-based models, graph traversal algorithms (e.g. depth-first search and random traversal) may be used to generate test cases that correspond to valid paths through the model. Memon [**48**] describes an Event Flow Graph (EFG) model for automated GUI testing. An EFG model represents a set of possible event sequences for a given AUT. The nodes represent GUI events and an edge from one event e_1 to another event e_2 indicates that e_2 can occur after e_1. Model-based testing techniques may also use state-based models to represent possible event sequences for a given AUT [**5, 51, 65**]. In state-based models, nodes represent GUI states and edges represent events that cause transitions between states.

Manual construction of behavioral models for automated GUI testing is a challenging task. Memon et al. [**47, 49**] and Nguyen et al. [**57**] describe the GUITAR framework for reverse engineering of GUI-based applications and automatic generation of event sequences. The GUITAR framework includes a GUI Ripper that automatically explores the GUI of the application under test to construct an EFG for model-based testing. Mesbah et al. [**51**] present a framework to automatically infer state machine models of web applications for mul-

tiple purposes including automated testing. Yang et al. [**73**] use static and dynamic analysis to automatically extract finite state machine models from mobile applications. Model-based testing techniques tend to generate infeasible test cases (i.e. test cases that are only partially executable) because abstract models may not accurately represent the actual behavior of the AUT.

Some existing research combines model-based testing and combinatorial-based techniques. Yuan et al. [**74**] use covering arrays and Event Interaction Graphs (EIG) to construct GUI event sequences that cover all t-way sequences of events. Wang et al. [**68**, **69**] use combinatorial techniques to automatically construct navigational graphs for web applications. They also describe a technique to test all pairwise (2-way) interactions between any two pages of a web application. Di Lucca et al. [**23**] present a technique that uses preexisting statechart models to test interactions between web applications and browsers. They use graph-based coverage criteria to generate test cases that cover all sequences of k transitions in the statechart model.

Online GUI testing techniques do not require a preexisting abstract model of the AUT prior to test generation. These techniques interact directly with the AUT to concurrently generate and execute test cases. Online GUI testing techniques use various strategies to determine which event to execute in each GUI state during event sequence generation. Event sequence generation is based on the runtime behavior of the AUT rather than traversal of abstract graph-based models. In comparison to model-based testing, online GUI testing techniques tend to achieve higher coverage of the AUT and are less likely to produce infeasible event sequences [**10**]. Carino [**17**] evaluates various online GUI testing algorithms for Java desktop applications [**60**]. The algorithms incrementally construct a graph-based model of previously executed event sequences and uses the model to guide subsequent execution of events. In this dissertation, we adapt some of the ideas described in [**17**] to Android apps. We develop online GUI testing algorithms that automatically generate replayable event sequences for a given Android application without need for static code analysis or construction of graph-based models.

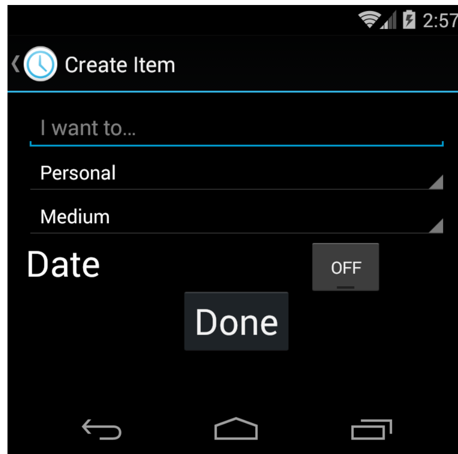


FIGURE 2.1. Example of an Android application GUI

2.5. Android Mobile Applications

Android mobile applications are Event Driven Software (EDS) and are composed of several Java components that are instantiated at runtime. An *Activity* is the primary GUI component of an Android application. Activities represent screens that are visible to users and each activity has a unique name within an application. Activities are composed of GUI widgets that users may interact with (e.g. buttons and text fields). Figure 2.1 shows an example of an activity with multiple widgets. Widgets in an activity are part of a hierarchical structure that defines a formal representation of an application's GUI at any point in time. Each widget has a set of properties and associated values (e.g. label, caption and size) that specifies its visual characteristics. Android apps often contain several activities in which user interaction may occur and only one activity can be active at any time.

Users often interact with Android apps by touching widgets on a screen or by performing actions on an input device (e.g. a keyboard or hardware button). The Android system may interact with activities to provide functionality that depends on external factors (e.g. network availability, screen orientation, battery levels, etc.). These interactions trigger GUI and system events that an Android application may respond to. Activities define *event handlers* that execute code in response to events. A sequence of events often causes transitions between activities or between states of the same activity. The Android system manages activities in a stack. Users can interact only with the activity at the top of the stack (i.e. the

running activity). When a user or system event launches a new activity, the *activity manager* adds the new activity to the stack. Android devices have "back" and "home" navigation buttons that are always available to users. In most Android apps, the "back" button removes the running activity from the activity stack and does one of the following: (i) reactivates the next activity in the stack or (ii) closes the Android app because the activity stack is empty. The "home" button always closes an Android app and returns to the device's home screen. In Chapter 3, we leverage our domain knowledge of Android applications to define a domain-specific metamodel that enables online construction of reusable test suites.

2.6. GUI Event Interaction in Android Applications

GUI-based Android applications are event-driven systems since they react primarily to GUI events. An event-driven system may exhibit a particular behavior only when two or more specific events are executed in the same sequence and in a particular order. Interactions between events that occur in a particular order may cause faulty behavior in an AUT [**36**]. Figure 2.2 shows a state transition graph from a real-world Android application (Tomdroid v1.7.2). The nodes represent GUI states and the edges represent events associated with particular widgets. The state transition graph has five GUI states $\{A, B, C, D, E, F, G, H\}$ and eight distinct events $\{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\}$. State A is the initial GUI state of the mobile application (i.e. the GUI state after a user launches the application). The widgets associated with each event are highlighted in rectangles. The two outgoing edges in state F represent the same event but with varying behavior depending on the path taken through the graph. We use this graph to illustrate Android application behavior that varies due to interaction between events that occur in a particular order.

Table 2.3 shows a test suite $T = \{t_1, t_2, t_3, t_4\}$ with four test cases derived from the state transition graph in Figure 2.2. Each test case t_i is an event sequence that represents a valid path through the state transition graph. Each event sequence covers a set of previously uncovered event pairs and results in a final GUI state. Test t_1 covers ten event pairs and results in GUI state F as shown in the graph. We construct t_2 by appending e_8 to t_1. Test t_2 covers five previously uncovered event pairs and results in GUI state G. At this point in

the test suite, no event pairs with e_4 and e_5 are covered because they do not appear in any of the event sequences.

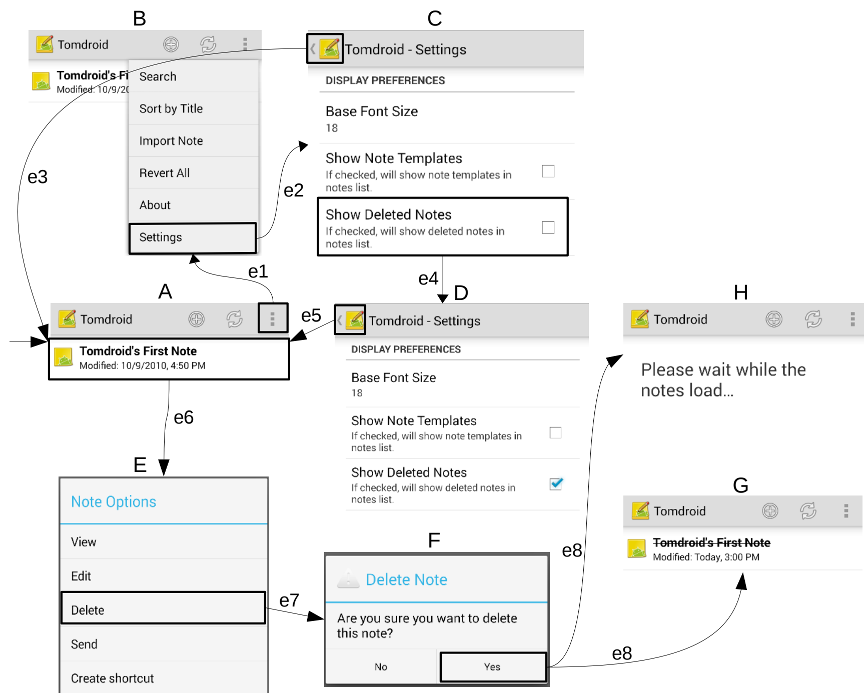


FIGURE 2.2. GUI state transition graph for a mobile application (Tomdroid)

ID	Start State	Event Sequence	New Pairs Covered	Final GUI State
t_1	A	$\langle e_1, e_2, e_3, e_6, e_7 \rangle$	$\{(e_1, e_2), (e_1, e_3), (e_1, e_6), (e_1, e_7), (e_2, e_3), (e_2, e_6), (e_2, e_7), (e_3, e_6), (e_3, e_7), (e_6, e_7)\}$	F
t_2	A	$\langle e_1, e_2, e_3, e_6, e_7, e_8 \rangle$	$\{(e_1, e_8), (e_2, e_8), (e_3, e_8), (e_6, e_8), (e_7, e_8)\}$	H
t_3	A	$\langle e_1, e_2, e_4, e_5, e_6, e_7 \rangle$	$\{(e_1, e_4), (e_1, e_5), (e_2, e_4), (e_2, e_5), (e_4, e_5), (e_4, e_6), (e_4, e_7), (e_5, e_6), (e_5, e_7)\}$	F
t_4	A	$\langle e_1, e_2, e_4, e_5, e_6, e_7, e_8 \rangle$	$\{(e_4, e_8), (e_5, e_8)\}$	G

TABLE 2.3. GUI event sequences and interaction-based behavior in a real-world Android app

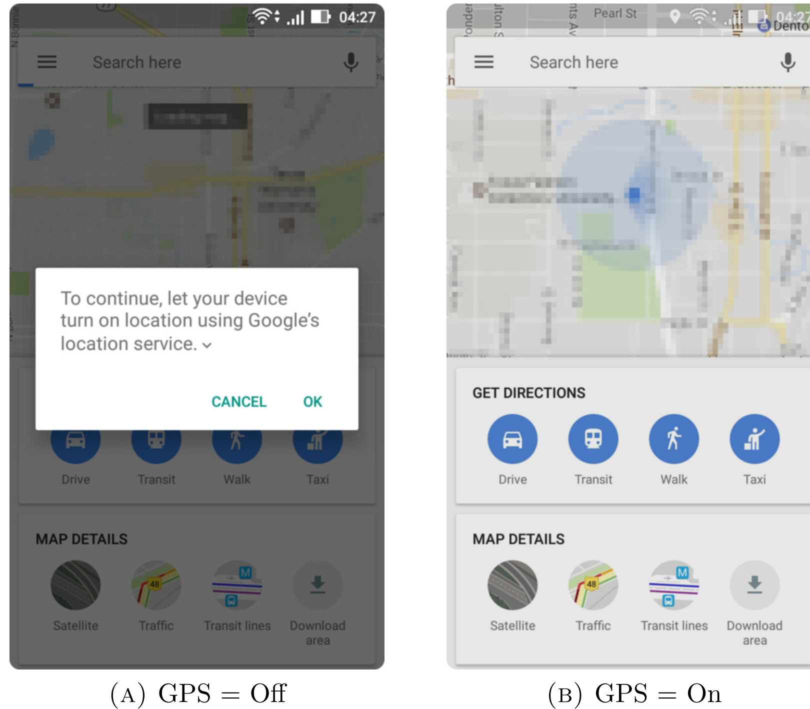
Consider the behavior of the mobile application when we construct event sequences with e_4 and e_5. Test t_3 takes the path through $\langle e_4, e_5 \rangle$ and covers nine new event pairs. This results in GUI state F. Test t_3 results in the same GUI state (F) as t_1 despite the presence of

e_4, e_5 and their corresponding pairwise interactions with other events in the sequence. This suggests that t_1 and t_3 cause identical behavior in the mobile application. It also suggests that appending e_8 to t_3 should result in the same GUI state as t_2, since t_2 is also the result of appending e_8 to t_1. However, we observe that t_4 results in GUI state H which represents new behavior that is not tested by t_1, t_2 and t_3. We cannot attribute this new behavior to the presence of e_8 in t_4 because e_8 is also in t_2. Test t_4 is unique because it covers two new event pairs, (e_4, e_8) and (e_5, e_8). An examination of Tomdroid's functionality shows that event e_4 configures the application to keep deleted notes rather than completely remove them from storage while e_8 confirms deletion of a note. The presence of e_4 and e_8 in t_4 such that e_4 precedes e_8 causes the note deletion event (e_8) to behave differently from other sequences that do not have this order relation between e_4 and e_8. We refer to this sort of behavior as *interaction-based* behavior.

A complex Android application may contain numerous instances where the order of GUI events affects its behavior. In Chapter 5, we describe a combinatorial-based technique that prioritizes coverage of n-way event combinations and increases the likelihood of testing behavior that occurs only when events are executed in a particular order.

2.7. Context-Sensitivity in Android Applications

A mobile application may use information from external sources (e.g. network devices, sensors, battery, the operating system, etc.) to provide context-sensitive functionality to users. These external sources of information define the operating context of a mobile application and may affect the behavior of an AUT. Figure 2.3 shows an example of context-sensitive behavior that occurs when a user launches a mobile application in different contexts (GPS on/off). If a user launches the application with the GPS turned off, the application displays a dialog that prompts the user to turn on the GPS sensor before proceeding. If a user launches the application with the GPS sensor turned on, the application retrieves the user's current location and does not display a dialog. This observation suggests that the behavior of an application may vary depending on its operating context and that it may be useful to generate tests that launch the AUT in different contexts.



(A) GPS = Off (B) GPS = On

Figure 2.3. Launching an Android application in two different contexts

Figures 2.4 and 2.5 show examples of context-sensitive behavior in an Android application when the same GUI event is executed in two different contexts. If a user clicks one of the list items in Figure 2.4 when the mobile device is in airplane mode, the application displays a dialog that informs the user about the absence of an Internet connection. In this case, the user is unable to access any other parts of the app beyond that point. Figure 2.5 shows what happens when the device is not in airplane mode and has access to the Internet. The application is able to retrieve the required information and display it to the user in a screen that may otherwise be unreachable in a different context. This observation suggests that it may be useful to generate tests that dynamically manipulate the operating context of the AUT to execute GUI events in different contexts.

The operating context of mobile applications changes constantly due to the portability and connectivity requirements of mobile devices. It is important for automated GUI testing techniques to consider the impact of changing context on the behavior of mobile applications. In Chapter 6, we describe a framework for automatic construction of test suites

that dynamically manipulate the operating context of the AUT to execute GUI events in different contexts and test context-sensitive behavior in Android applications.

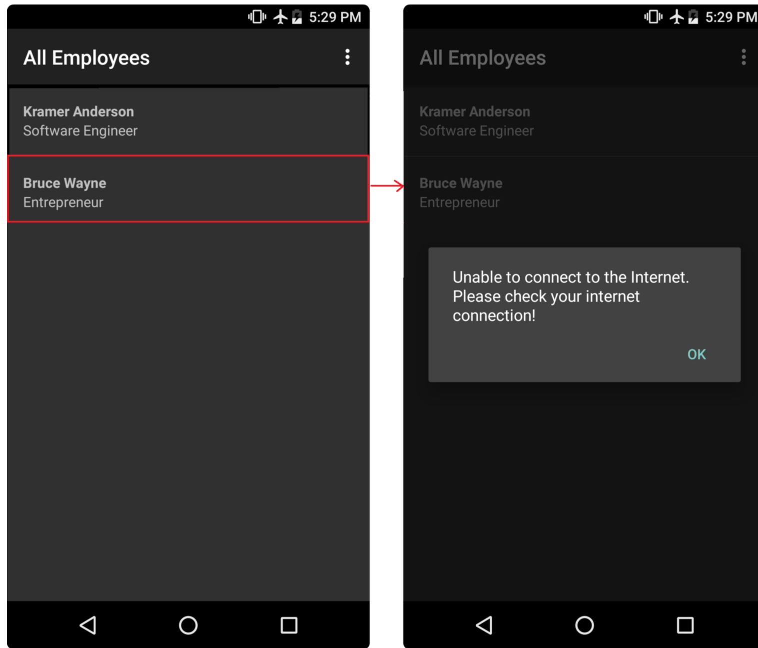


FIGURE 2.4. Clicking a list item with airplane mode ON (no internet access)

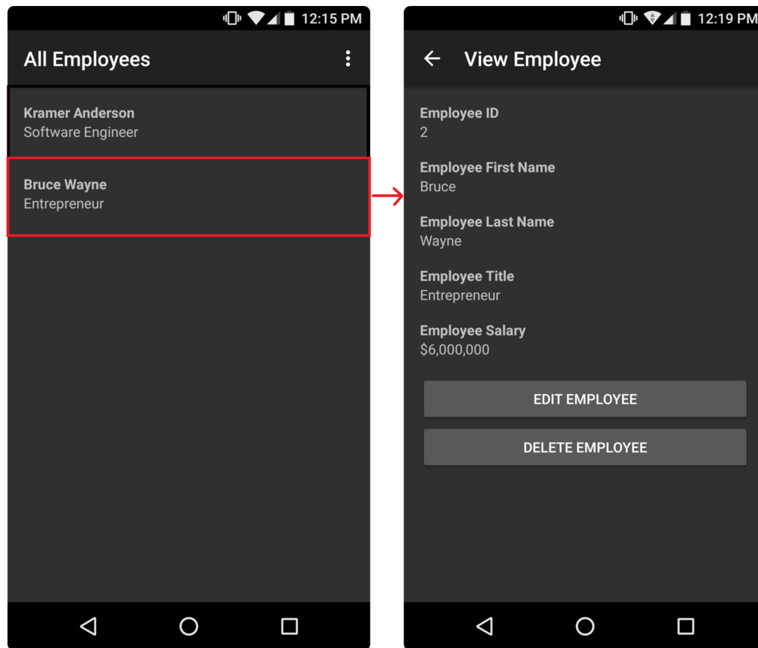


FIGURE 2.5. Clicking a list item with airplane mode OFF (internet access)

2.8. Automated GUI Testing of Android Applications

The majority of existing research in automated testing of Android applications explores model-based techniques and pays limited attention to online GUI testing techniques. Many of these techniques do not consider the existence of interactions among events executed in a particular order and they do not consider the potential impact of context changes on the behavior of mobile applications.

There are several tools and techniques for automated GUI testing of Android applications [**2, 5, 28, 29, 32, 43, 53, 55, 56, 62, 64, 65, 77, 78**]. Many of the tools developed in prior research studies are no longer compatible with recent versions of the Android operating system. This makes it difficult to use the tools for further research in automated GUI testing of Android apps. Monkey [**29**] is the most widely available automated GUI testing tool for Android applications since it is part of the official Android Software Development Kit (SDK) and is particularly easy to use [**19**]. Monkey automatically executes a predefined number of pseudorandom GUI and system events on any given Android application. It performs actions on random screen coordinates regardless of whether the events are relevant to the application under test. We do not consider Monkey to be an online GUI testing tool since it does not use any runtime information concerning the GUI structure of the application under test. Amalfitano et al. [**5**] develop a model-based testing tool called MobiGUITAR that automatically extracts a state machine model and traverses the model to automatically generate event sequences. Machiry et al. [**42**] develop a tool called Dynodroid that uses online GUI testing techniques to execute an input sequence of predefined length for any given Android app. Dynodroid does not provide a way to construct a test suite with distinct event sequences that are replayable. It does not consider potential interactions between events as part of its input generation process and provides limited consideration for events that change the operating context of the AUT. Nguyen et al. [**58**] describe a technique that combines model-based testing and combinatorial testing to generate event sequences from a manually constructed finite state model. Their technique converts event sequences into concrete test cases with combinatorial input data. The technique does not consider the order of events or

coverage of event combinations. Trimdroid [**53**] is a framework for GUI testing of Android applications that uses combinatorial-based methods, automated program analysis and formal specifications to generate tests. Trimdroid extracts models from application source code and uses graph-based criteria to generate event sequences that are enhanced with combinatorial input data. It analyzes source code to detect dependencies between GUI elements and uses the derived information to reduce the number of input combinations to be tested.

This dissertation describes novel algorithms and heuristics for online construction of Android application test suites. We use an event sequence metamodel to specify information that each test case must contain to enable reexecution during regression testing. We also use our event sequence model to specify equivalence relations between events. Our online algorithms use the equivalence relations to guide event sequence generation toward coverage of events and event combinations. In Chapter 3, we describe our automated GUI testing tool, Autodroid, that uses online GUI testing techniques to generate reusable test suites for Android applications without need for source code analysis or graph-based models.

CHAPTER 3

ONLINE CONSTRUCTION OF REUSABLE TEST SUITES FOR ANDROID APPLICATIONS

Model-based techniques generate tests from a preexisting abstract model of the Application Under Test (AUT) and may produce infeasible test cases. Online GUI testing techniques reduce the likelihood of infeasible tests and often achieve higher code coverage than model-based techniques [**10**]. Online GUI testing algorithms (also known as dynamic event extraction-based algorithms) interact directly with the GUI of the application under test to concurrently generate and execute event sequences without need for source code analysis or preexisting abstract models of the AUT. These algorithms iteratively identify, select and execute events to generate tests one-event-at-a-time [**3**]. In this chapter, we describe an online approach to automatic construction of Android application test suites. We describe an event sequence metamodel that specifies information that each test case must contain to enable reexecution for automated regression testing and failure reproduction. We use the event sequence metamodel to define equivalence relations between events and we develop an algorithm for online construction of test suites with distinct event sequences that conform to our event sequence metamodel. We implement our techniques in a tool called Autodroid and compare test suites generated with our online technique to test suites generated with Monkey [**29**], a widely available random GUI testing tool for Android applications.

Autodroid and the online techniques in this chapter differ from related work. First, many existing tools and techniques require a preexisting abstract model of the Application Under Test (AUT) for *offline* event sequence generation and subsequent execution [**4**, **5**, **43**, **77**]. Our online algorithms interleave generation and execution of event sequences without need for source code analysis or preexisting abstract models of the AUT. Second, in a comparison of automated testing tools for Android applications, Choudhary et al.[**19**] notes that the majority of existing tools do not produce event sequences that can be reexecuted for regression testing purposes or reproduction of failures. These tools do not produce

a structured representation of the event sequences that can be replayed. Many existing tools automatically execute a single event sequence of predetermined length for the entire duration of testing. It may be difficult for a tester to reexecute and inspect a single sequence that contains a large number of events, especially when such events do not conform to a predefined abstract representation. Our automated GUI testing tool, Autodroid, generates test suites with distinct event sequences of varying length and our test suites conform to an event sequence metamodel that enables reuse. Test suites with distinct event sequences of varying length may be easier to inspect since testers may need to identify and examine only the particular test cases that fail without having to examine any other test cases in the test suite. Short event sequences may identify "shallow" faults that are easy to reproduce. Long event sequences may improve code coverage and identify faults that short event sequences cannot reach [**71**]. Finally, many existing tools are no longer compatible with recent versions of the Android operating system (Android 4.0 and above). This limits the ability of researchers to use these tools as the basis for further research in automated GUI testing. Autodroid is compatible with recent versions of the Android operating system and provides tool support for the empirical studies in subsequent chapters of this dissertation.

3.1. An Event Sequence Metamodel for Online GUI Testing

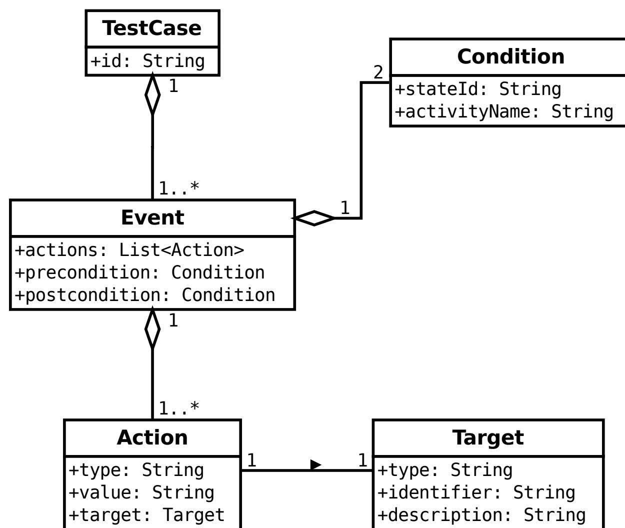


Figure 3.1. GUI event sequence metamodel

Figure 3.1 shows our event sequence metamodel for online construction of reusable Android application test suites. We define each element of the metamodel as follows.

Definition 3.1. (Target) A target is a GUI widget or other non-GUI component that a user or the Android system may interact with. A target has an identifier, type and description. The *identifier* is a name that uniquely identifies a GUI widget or non-GUI component e.g. "btnSignUp" or "wifiDevice". The *type* attribute denotes what kind of GUI widget or non-GUI component the target represents e.g. "Button", "TextBox" or "gps". The *description* attribute describes the visual characteristics of a target whenever possible. For example, a button may be visually described by its label. Examples of targets include buttons, checkboxes, GPS and WiFi .

We define an equivalence relation between targets as follows: *two targets are equivalent if they have the same type, identifier and description.*

Definition 3.2. (Action) An *action* characterizes a user or system interaction with a *target*. Every action has a *type* and *value*. An action's *type* denotes the nature of the interaction (e.g. "click" and "text entry"). Some actions have an associated *value* e.g. the specific text value entered in a text field. An action's *target* represents the GUI widget or non-GUI component that the action affects. Users execute *user actions* via the GUI of an app (e.g. clicking a specific button) and the Android system may execute *system actions* (e.g. switching GPS off).

We define an equivalence relation between actions as follows: *two or more actions are equivalent if they have the same type and target.*

Definition 3.3. (Event) An event is a *sequence of actions* that has a *precondition* and *postcondition*. Event execution occurs when a user or automated tool performs all the actions specified in an event's action sequence. An event's precondition describes the GUI state of the application under test prior to execution of the event. An event's postcondition describes the GUI state of the application under test after execution of the event.

We define an equivalence relation between events as follows: *two or more events are*

equivalent if they have the same precondition and sequence of actions.

Note that the equivalence relation between events does not include postconditions. In subsequent chapters, we use equivalence relations between events to distinguish between previously executed events and events that are yet to be executed. During online test generation, the postcondition of an event is unknown prior to execution of the event. We refer to events with undefined postconditions as *partial events*. An event with a defined precondition, action sequence and postcondition is called a *complete event*.

The state of a mobile application's GUI often changes after execution of one or more events. An event may consist of one or more actions. We refer to events with only one action as *simple events* while events that have more than one action are *complex events*. An event that closes the application under test is called a *termination event*.

Definition 3.4. (Preconditions and Postconditions) The pre- and postconditions of an event describe the GUI state of an AUT prior to and after execution of the event respectively. Each precondition and postcondition has an *activityName* and a *stateId*. The *activityName* uniquely identifies the *running activity* of an AUT prior to and after event execution. The *stateId* is a unique identifier for GUI states before and after event execution. In this work, we derive the *stateId* from the name of the running activity and the set of available actions on a given screen. *Two GUI states are equivalent if they have the same activity name and equal sets of available actions.*

Postconditions and preconditions enable an automated regression testing tool to ensure that an Android app is in the expected GUI state before and after execution of each event in a test case.

Definition 3.5. (Test case) A *test case* is a sequence of events. The length of a test case is the number of events in the sequence. Every test case has an *id* that uniquely identifies it within a test suite. Table 3.1 shows an example of a test case defined in terms of our event sequence model.

Definition 3.6. (Test suite) A test suite is a set of test cases.

Start ID	`<testcase>` `<id>testcase0001</id>`

Event 1	`<event>`
	`<precondition>`
	`<activityName>RegActivity</activityName>`
	`<stateId>e9be7ae186ac52a8dcc</stateId>`
	`</precondition>`
	`<actions>`
	`<action>`
	`<type>click</type>`
	`<value></value>`
	`<target>`
	`<id>btn_next</id>`
	`<type>Button</type>`
	`<desc>Next</desc>`
	`</target>`
	`</action>`
	`</actions>`
	`<postcondition>`
	`<activityName>BioActivity</activityName>`
	`<stateId>5336ab0c86f2c254de4</stateId>`
	`</postcondition>`
	`</event>`

Event 2	`<event>`
	`<precondition>`
	`<activityName>BioActivity</activityName>`
	`<stateId>5336ab0c86f2c254de4</stateId>`
	`</precondition>`
	`<actions>`
	`<action>`
	`<type>click</type>`
	`<value></value>`
	`<target>`
	`<id>btn_finish</id>`
	`<type>Button</type>`
	`<desc>Finish</desc>`
	`</target>`
	`</action>`
	`</actions>`
	`<postcondition>`
	`<activityName>finishActivity</activityName>`
	`<stateId>2447bd0f37c2e791fa2</stateId>`
	`</postcondition>`
	`</event>`

End	`</testcase>`

Table 3.1. Example test case with two events

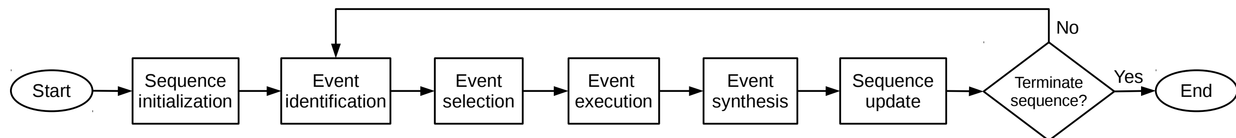


FIGURE 3.2. Online event sequence construction

Figure 3.2 shows the steps in our online approach to event sequence generation. The steps are as follows:

Step 1: Sequence Initialization. This step creates an empty sequence, deletes application data created by previously executed sequences (if any) and launches the AUT on the mobile device. This step produces the initial GUI state of the application under test.

Step 2: Event Identification. This step analyzes the GUI of the application under test, identifies all available actions and extracts a representation of the current GUI state to be used as a precondition for each available event. We use this information to derive a set of event abstractions that conform to our event sequence metamodel. At this point, the postconditions for the identified events are unknown. The output of this step is a set of partial events that can be executed from the current GUI state.

Step 3: Event Selection. This step uses an event selection strategy to choose an event from the set of events identified in step 2.

Step 4: Event Execution. This step executes the event selected in step 3. This is done via direct interaction with the GUI of the application under test. Event execution often causes the AUT to change its GUI state.

Step 5: Event Synthesis. This step observes the AUT's response to event execution, extracts the postconditions of the executed event and updates the appropriate event abstraction. The output of this step is a complete event with preconditions, actions and postconditions.

Step 6: Sequence Update. This step adds the executed and synthesized event to the event sequence under construction.

Our online technique iteratively identifies, selects and execute events (steps 2–6) to

incrementally construct an event sequence one-event-at-a-time. The objective is to execute and extract event abstractions that conform to our event sequence metamodel. We refer to this iterative process as an *event extraction cycle*. The event sequence generation process ends when a predefined termination criterion is met.

Algorithm 1: Random-based test suite construction

```
   Input  : application under test, AUT
   Input  : test case termination criterion, t_end
   Input  : test suite completion criterion, T_comp
   Output: test suite, T
 1 T ← φ                                          ▷ set of event sequences (test suite)
 2 E_term ← φ                                     ▷ set of termination events to avoid
 3 while test suite completion criterion is not satisfied do
 4 |  clear application data and start AUT
 5 |  t_i ← φ                                     ▷ event sequence (test case)
 6 |  repeat
 7 |  |  E_all ← getAvailableEvents()
 8 |  |  E_all ← removeTerminationEvents(E_all, E_term)
 9 |  |  e_sel ← selectRandomEvent(E_all)
10 |  |  execute and synthesize event e_sel
11 |  |  if e_sel closed the AUT then
12 |  |  |  E_term ← E_term ∪ {e_sel}              ▷ update set of termination events
13 |  |  end
14 |  |  t_i ← t_i ∪ {e_sel}                       ▷ sequence update
15 |  until test case termination criterion is satisfied
16 |  T ← T ∪ t_i                                 ▷ add event sequence to test suite
17 end
```

3.3. Random-based Test Suite Construction

Algorithm 1 shows pseudocode for online construction of test suites with distinct event sequences. The algorithm uses the iterative process in Figure 3.2 to construct multiple event sequences that each represent a single test case. It requires the following input: (i) the application under test (ii) a test case termination criterion and (iii) a test suite completion criterion. The test case termination criterion specifies when to terminate each event sequence and the test suite completion criterion specifies when a test suite is complete. The test case

termination criterion may be a specified number of events or some other randomized criterion. The test suite completion criterion could be a specified number of test cases or a fixed time limit.

The test suite construction algorithm consists of an event extraction cycle that iteratively selects and executes GUI events uniformly at random. Thus, we refer to the algorithm as *random-based*. The algorithm maintains a set of termination events E_{term} that it uses to avoid selection of any previously executed event that explores beyond the boundaries of the AUT (e.g. events associated with the "back" button or an "exit" button). Lines 4-5 initialize an empty sequence at the start of each test case and clears any application data generated by previous event sequences. This initialization process ensures that event sequences in a test suite are independent of one another and can be reexecuted in isolation. Lines 3-17 represent the event extraction cycle that incrementally constructs each event sequence. The *getAvailableEvents* procedure call on line 7 identifies the set of available events E_{all} in each GUI state and creates corresponding abstractions for each event. The *removeTerminationEvents* procedure call on line 8 removes any known termination events from E_{all} to encourage exploration of the AUT. The *selectRandomEvent* procedure call on line 9 selects an event from the set of available events E_{all} uniformly at random. Line 10 executes the selected event and updates the event abstraction with the appropriate postconditions. If the executed event closes the AUT, line 12 updates the set of termination events E_{term} to ensure that the event is excluded from subsequent interations of the event extraction cycle. Line 14 updates the event sequence at the end of each event extraction cycle. Line 16 adds the generated event sequence to the test suite at the end of each test case.

Test case termination. Our objective is to generate test suites that contain distinct event sequences of varying length. To achieve this objective, we define a test case termination criterion that uses a predefined probability value to pseudorandomly terminate each event sequence. Our test case termination criterion only terminates a test case when such termination will not produce a duplicate test case in the test suite under construction. The algorithm also guarantees termination of a test case when an event explores beyond the boundaries of

the AUT. Algorithm 2 shows pseudocode for our test case termination criterion. The algorithm requires the following as input: (i) the termination probability, $0 < p < 1$ (ii) the test case under construction t_i and (iii) the test suite under construction, T. The *random(0,1)* procedure call on line 1 pseudorandomly generates a real number between 0 and 1. We use this termination criterion in subsequent chapters of this dissertation.

Algorithm 2: Probabilistic criterion for termination of event sequences

Input : termination probability, $0 < p < 1$
Input : test case under construction, t_i
Input : test suite, T

```
1 if (random(0,1) < p and t_i ∉ T) or AUT is closed then
2 |   return true
3 end
4 return false
```

3.4. Tool Implementation

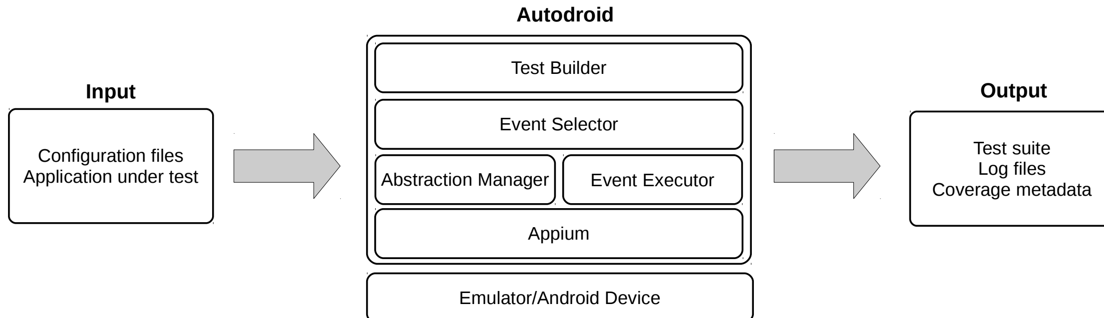


FIGURE 3.3. Input, output and major components of Autodroid

We implement our algorithms in our automated GUI testing tool, Autodroid. Autodroid uses our random-based algorithm to automatically construct Android application test suites with event sequences that conform to our metamodel. The event data in each sequence enables Autodroid to reexecute previously generated test suites and perform GUI-level assertions to verify preconditions and postconditions. Figure 3.3 shows the input, output and major components that Autodroid uses for online test suite construction. Autodroid takes an Android application package file (APK) and a configuration file as input. The configuration

file allows users to specify a test case termination criterion and test suite completion criterion. Autodroid is implemented in Java and it uses the Appium test automation library[1] to interact with the GUI of Android applications. The major components of Autodroid include a test builder, event selector, event executor and abstraction manager.

Test builder. The test builder initiates and coordinates each step of the event sequence generation process. It performs event sequence updates in each iteration of the event extraction cycle and maintains a cache of generated event sequences.

Abstraction manager. The abstraction manager uses the Appium library to identify events in GUI states and create event abstractions that conform to our metamodel. The abstraction manager identifies the preconditions and postconditions of events and updates event abstractions as necessary during the test generation process.

Event selector. In each iteration of the event extraction cycle, the event selector chooses an event to execute from the set of available events in the AUT's current GUI state.

Event executor. The event executor receives an abstract representation of an event and uses the provided information to execute the event on the AUT.

In each iteration of the event extraction cycle, Autodroid always includes an event associated with the "back" navigation button in the set of available events and excludes the "home" navigation button. Whenever Autodroid encounters a GUI state with text input widgets, it generates and executes a complex event that fills out each text input widget with random strings before interacting with a non-text widget (e.g. a button).

3.5. Experiments

We use Monkey as a baseline for evaluation of test suites generated with Autodroid. Monkey is one of the most widely used tools for automated GUI testing of Android applications since it is part of the official Android developer toolkit. It is also one of the few automated GUI testing tools that remains compatible with recent versions of the Android operating system. We perform experiments with seven Android applications to answer the following research question: **does our online random-based technique (Autodroid)**

[1]http://appium.io

generate Android application test suites that achieve higher code coverage than test suites generated with Monkey?

3.5.1. Subject Applications

App Name	**Lines**	**Methods**	**Classes**	**Activities**
Tomdroid v0.7.2	5,736	496	131	8
Loaned v1.0.2	2,837	258	70	4
Budget v4.0	3,159	367	67	8
A Time Tracker v0.23	1,980	130	22	5
Repay v1.6	2,059	204	48	6
Moneybalance v1.0	1,460	163	37	5
WhoHasMyStuff v1.0.25	1,026	90	24	2

Table 3.2. Characteristics of selected Android apps

We perform experiments with seven Android applications randomly selected from the F-droid app repository[2]. We exclude games and system services with no GUIs from our experiments and only consider apps that allow automatic bytecode instrumentation with the techniques described in Zhauniarovich et al. [**78**]. We instrument the bytecode of each subject application to collect code coverage measurements for our study. Table 3.2 shows characteristics of the subject applications. The applications range from 1,026 to 5,736 source lines of code (SLOC) and 3,597 to 22,169 blocks of bytecode.

3.5.2. Experimental Setup

Monkey and Autodroid are different in one critical aspect. Monkey executes a single sequence of pseudorandom events up to a user-specified length. Autodroid generates test suites with multiple event sequences of varying length. We run multiple executions of Monkey for each subject application to simulate multiple test cases within a test suite. We configure Monkey to generate multiple event sequences of length $n = 192$, where n is the number of events in the longest test case generated with Autodroid across all subject applications. Therefore, each event sequence generated with Autodroid has a smaller or equal number of

[2]https://f-droid.org

events as the corresponding event sequence generated with Monkey (i.e. length ≤ 192). We use a 5% probability value to pseudorandomly terminate event sequences in Autodroid.

We perform our experiments on Android 4.4.4 emulator instances configured with 4 processors and 2GB RAM. We use each tool to generate 10 test suites for each subject application with a fixed time limit of two hours (120 minutes) for each test suite. We do not set a time delay between execution of events in Monkey since the tool does not require time to analyze the GUI of the application under test. In Autodroid, we specify a two-second delay between execution of consecutive events to enable the AUT respond to each event before extraction of event data.

3.5.3. Results

Application	**Code coverage (%)**	
	Monkey	**Autodroid**
Tomdroid	33.22	**45.11**
Loaned	20.89	**53.53**
Budget	58.60	**66.06**
A Time Tracker	38.24	**70.58**
Repay	16.87	**47.75**
Moneybalance	21.81	**75.51**
WhoHasMyStuff	70.38	**75.59**

Table 3.3. Mean block coverage for Monkey and Autodroid across 10 test suites for each subject application

Table 3.3 shows the mean block coverage across 10 test suites for each subject application. The results show that Autodroid generates test suites that achieve 5%-53% higher block coverage compared to the test suites generated with Monkey. Figure 3.4 shows the distribution of block coverage values across 10 test suites generated with each tool for each subject application. The results show that the test suites generated with Autodroid have higher median block coverage in all seven subject applications compared to the median block coverage of test suites generated with Monkey. The maximum block coverage achieved for each subject application is higher for test suites generated with Autodroid compared to test suites generated with Monkey.

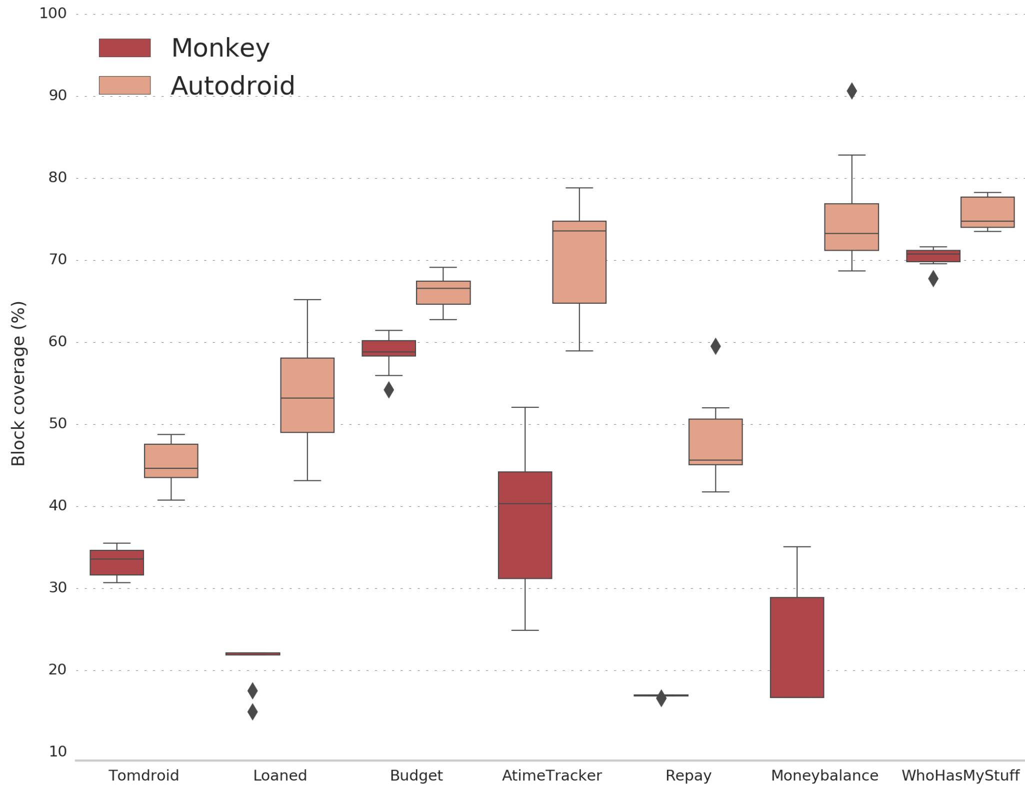


FIGURE 3.4. Distribution of block coverage values across 10 suites generated with Autodroid and Monkey for each subject application

3.5.4. Discussion and Implications

Autodroid generates test suites that achieve higher block coverage than those generated with Monkey. In each iteration of its event extraction cycle, Autodroid analyzes the GUI of the application under test to determine the set of available events that can be executed in each GUI state. Monkey executes a large number of irrelevant events that do not contribute to code coverage because it does not analyze the GUI of the application under test during test generation. This limits Monkey's ability to explore the AUT given a limited number of events to execute in each test case. Autodroid generates test suites with fewer events than those generated with Monkey and achieves higher code coverage since it limits event execution to relevant events.

Monkey is designed to generate and execute a large number of pseudorandom events in a single sequence. This makes Monkey best suited for scenarios that do not require

reusable test cases and easy reproduction of event sequences that expose faults. We designed Autodroid to automatically generate test suites with distinct event sequences (test cases) of varying length, such that each sequence can be reexecuted in isolation. Unlike Monkey, Autodroid extracts an abstract representation of each event sequence in a test suite. This abstract representation contains information required to automatically reexecute event sequences with GUI-level assertions during regression testing or failure reproduction. The ability to automatically replay event sequences may make it easier for testers to understand and reproduce failure-causing test cases.

3.5.5. Threats to Validity

The primary threat to the validity of our empirical study is the randomized nature of the algorithms and tools used in our experiments. For each app, we generate 10 test suites with each tool to minimize this threat. We use a 5% probability value for the pseudorandom test termination criterion in Autodroid. Different probability values may produce different results.

3.6. Summary and Conclusions

In this chapter, we describe an event sequence metamodel that enables automatic construction of reexecutable test suites for Android applications. We describe an online process to automatically generate event sequences that conform to our metamodel and we develop algorithms and heuristics to automatically construct Android application test suites with distinct event sequences of varying length. We demonstrate the feasiblity of our techniques with an automated GUI testing tool, *Autodroid*. We perform experiments with seven Android applications and show that Autodroid generates test suites that achieve higher code coverage compared to test suites generated with Monkey, a random GUI testing tool that is part of the Android developer toolkit.

In Chapter 4, we describe probabilistic event selection strategies to improve the code and event coverage of test suites generated with online techniques.

CHAPTER 4

FREQUENCY-BASED TEST SUITE CONSTRUCTION

In Chapter 3, we described an online test suite construction algorithm that iteratively selects and executes events uniformly at random to construct each test case. We refer to the algorithm as *random-based* to denote its use of uniform random event selection in each GUI state. Event selection is a key step in online event sequence generation and the strategy used to choose which event to execute may affect the quality of the resulting test suites. During construction of a particular test case t_i in test suite T, an algorithm that selects events uniformly at random has no mechanisms to avoid redundant selection of an event e_j that has already been executed in t_i or in a previous test case $t_{(i-k)} \in T$. Redundant execution of events may inhibit code and event coverage especially when it occurs in GUI states with events that have not yet been executed in the test suite. Consequently, random-based algorithms may fail to explore parts of the GUI that cover significant amounts of code or expose faults. While there is a significant body of work on automated GUI testing tools and model-based techniques [**5**,**10**,**32**,**41**,**42**,**46**,**48**,**48**,**53**,**64**,**65**,**77**,**78**], prior research gives little attention to event selection strategies for online construction of Android application test suites.

In this chapter[1], we extend our random-based test suite construction algorithm to include a cache of previously executed events and their prior execution frequencies in the test suite under construction. We use equivalence relations between events (as defined in chapter 3) to distinguish between events and determine whether or not they have been previously executed. We develop two frequency-based event selection strategies that use the prior execution frequencies of each available event in a GUI state to dynamically alter event selection probabilities. During test suite construction, our frequency-based event selection algorithms prioritize selection of events that have not yet been previously executed in the

[1]Parts of this chapter have been previously published in D. Adamo, R. Bryce, T.M. King. Randomized Event Sequence Generation Strategies for Automated Testing of Android Apps. In *Information Technology - New Generations*, pp. 571-578. Springer, Cham, 2017.

test suite under construction. The objective is to use frequency information about previously executed events to minimize redundant event execution and maximize code coverage and event coverage in each test case.

Algorithm 3: Frequency-based test suite construction

Input : application under test, AUT
Input : test case termination criterion, t_{end}
Input : test suite completion criterion, T_{comp}
Output: test suite, T

1. $T \leftarrow \phi$ ▷ `set of event sequences (test suite)`
2. $F_{map} \leftarrow$ map of distinct events to prior execution frequencies
3. $E_{term} \leftarrow \phi$ ▷ `set of termination events to avoid`
4. **while** *test suite completion criteria not true* **do**
5. clear application data and start AUT
6. $t_i \leftarrow \phi$ ▷ `event sequence (test case)`
7. **repeat**
8. $E_{all} \leftarrow$ getAvailableEvents()
9. $E_{all} \leftarrow$ removeTerminationEvents(E_{all}, E_{term})
10. $e_{sel} \leftarrow$ selectEvent(E_{all}, F_{map})
11. execute and synthesize event e_{sel}
12. **if** *application is closed* **then**
13. $E_{term} \leftarrow E_{term} \cup \{e_{sel}\}$ ▷ `add to set of termination events`
14. **end**
15. $t_i \leftarrow t_i \cup \{e_{sel}\}$
16. update selection frequency of e_{sel} in F_{map}
17. **until** *test case termination criteria is true*
18. $T \leftarrow T \cup t_i$
19. **end**

4.1. Test Suite Construction Algorithm

Algorithm 3 shows our modified algorithm for online construction of Android application test suites. The defining characteristic of the algorithm is that it maintains a history of prior event execution frequencies and uses the frequency information to choose which event to execute. Thus, we refer to the algorithm as *frequency-based*. The algorithm takes the following input: (i) the application under test (ii) a test case termination criterion and (iii) a test suite completion criterion. The test case termination criterion specifies when to

terminate each event sequence and the test suite completion criterion specifies when a test suite is complete. The test case termination criterion may be a specified number of events or some other randomized criterion. The test suite completion criterion could be a specified number of test cases or a fixed time limit.

The frequency-based test suite construction algorithm uses the same event extraction cycle (lines 7-17) as the random-based algorithm we described in chapter 3 with the addition of a step (line 16) that updates the prior execution frequency of each executed event. It begins with an empty test suite (line 1) and generates multiple event sequences as test cases. It terminates event sequences when the specified test case termination criterion is satisfied and ends the test suite construction process when the specified test suite completion criterion is satisfied. Line 2 initializes an empty map of previously identified events and their prior execution frequencies in the test suite. The prior execution frequency of any identified event that has never been executed is zero. The algorithm maintains a set of termination events E_{term} that it uses to avoid selection of any previously executed event that explores beyond the boundaries of the AUT (e.g. events associated with the "back" button or an "exit" button). Line 16 updates the map of events and execution frequencies each time the algorithm executes an event.

The implementation of the *selectEvent* function call on line 10 defines the event selection strategy used to choose which event to execute in each GUI state. A uniform random strategy selects an event uniformly at random in each GUI state without considering the prior execution frequency of each event. In uniform random event selection, each event in a GUI state is equally likely to be selected and the probability distribution never changes. Uniform random selection is often implemented with pseudorandom number generators that select a random event from the set of available events in each GUI state. In this chapter, we present frequency-based alternatives to uniform random selection.

4.1.1. Frequency Weighted Event Selection

This strategy computes the event selection probability of each available event based on the number of times the event has been previously executed in the test suite under

construction. Similar to uniform random selection, each event in a GUI state may be selected. Unlike uniform random selection, every event in a GUI state is *not equally likely* to be selected.

Algorithm 4: Frequency weighted event selection algorithm

```
   Input  : set of available events in GUI state, E_all
   Input  : map of events and prior execution frequencies, F_map
   Output: selected event, e_sel
 1 function freqWeightedSelection(E_all, t_i, F_map)
 2     totalWeight ← 0.0
 3     for event in E_all do
 4         totalWeight ← totalWeight + getWeight(event)
 5     end
 6     e_sel ← first event in E_all
 7     selectionWeight ← random(0,1) × totalWeight
 8     weightCount ← 0.0
 9     for event in E_all do
10         weightCount ← weightCount + getWeight(event)
11         if weightCount ≥ selectionWeight then
12             e_sel ← event
13             return e_sel
14         end
15     end
16     return e_sel
```

Algorithm 4 shows the frequency weighted selection algorithm. The algorithm takes the set of available events in a GUI state as input and assigns weights to each event based on prior execution frequency. The *random(0,1)* procedure call on line 7 generates a pseudorandom real number between 0 and 1. The weight of each event in a GUI state is given by:

$$weight(e) = \frac{1}{N(e) + 1} \qquad (1)$$

where e is an event and $N(e)$ is the number of times the event has been previously executed in the test suite. The algorithm makes a pseudorandom selection biased by the weight of each

available event. In any given GUI state with a set of available events E_{all}, the algorithm is more likely to select an event $e_i \in E_{all}$ that has been previously executed fewer times relative to other available events. Note that every event in E_{all} has a non-zero probability of selection, but the probability varies across events.

4.1.2. Minimum Frequency Event Selection

This strategy considers only the subset of available events that have been executed least frequently in a given GUI state. Unlike uniform random and frequency weighted selection, there are instances in which some events in a GUI state have no chance of selection. This strategy gives exclusive consideration to the least frequently executed events in a GUI state.

Algorithm 5: Minimum frequency event selection algorithm

Input : set of available events in GUI state, E_{all}
Input : map of events and prior execution frequencies, F_{map}
Output: selected event, e_{sel}

```
1  function minFreqSelection(E_all, F_map)
2      candidates ← φ
3      minFreq ← ∞
4      for event in E_all do
5          selectionFreq ← getExecutionFrequency(event)
6          if selectionFreq < minFreq then
7              candidates ← φ
8              candidates ← candidates ∪ {event}
9              minFreq ← selectionFreq
10         else if selectionFreq = minFreq then
11             candidates ← candidates ∪ {event}
12         end
13     end
14     e_sel ← selectRandom(candidates)
15     return e_sel
```

Algorithm 5 shows the minimum frequency selection algorithm. The algorithm takes the set of available events in a GUI state as input. It iterates through the set of available events and identifies the subset of events that have been executed the least number of times.

All events that are not in this subset are discarded. If there is more than one event that has been previously executed the least number of times, the algorithm makes a uniform random selection (i.e. random tie breaking). This event selection strategy is a non-deterministic variant of the *Frequency* strategy described in Machiry et al. [**42**].

4.2. Implementation

We extend our automated GUI testing tool, Autodroid, to include a *frequency engine* that keeps track of the prior execution frequency of each distinct event. Autodroid's frequency engine and event selector use the frequency information to dynamically alter event selection probabilities in each iteration of the event extraction cycle.

4.3. Experiments

4.3.1. Research Questions

We perform an empirical study with seven Android applications to address the following research questions:

RQ1: Do the frequency-based event selection strategies generate test suites that achieve higher code coverage than those generated with uniform random event selection?

RQ2: Do the frequency-based event selection strategies generate test suites that achieve higher event coverage than those generated with uniform random event selection?

4.3.2. Subject Applications

App Name	**Lines**	**Methods**	**Classes**	**Activities**
Tomdroid v0.7.2	5,736	496	131	8
Loaned v1.0.2	2,837	258	70	4
Budget v4.0	3,159	367	67	8
A Time Tracker v0.23	1,980	130	22	5
Repay v1.6	2,059	204	48	6
Moneybalance v1.0	1,460	163	37	5
WhoHasMyStuff v1.0.25	1,026	90	24	2

TABLE 4.1. Characteristics of selected Android apps

We evaluate each event selection strategy on seven real-world Android applications. Each application is publicly available in the F-droid app repository[2] and/or Google Play Store[3]. Table 4.1 shows the characteristics of the selected applications. The apps range from 1,026 to 5,736 source lines of code (SLOC), 90 to 496 methods, 24 to 131 classes and 2 to 8 activities. Since our implementation relies on Android GUI testing libraries, we select apps with GUIs that predominantly use the standard widgets provided by the Android framework. We limit our selection of apps to those that allow automatic bytecode instrumentation [**78**] without direct modification of source code.

4.3.3. Experimental Setup

Our experiments examine the following event selection strategies: (i) *Rand* – uniform random event selection (ii) *FreqWeighted* – frequency weighted event selection and (iii) *MinFrequency* – minimum frequency event selection.

We perform our experiments on Android 4.4 emulator instances with 4 processors and 2GB RAM. For each subject application, we generate 10 test suites using each event selection strategy (uniform random, frequency weighted and minimum frequency). We use a fixed time budget of two hours (120 minutes) to generate each test suite. We set a two-second delay between execution of consecutive events in each test case to give the AUT time to respond to each event. We use a fixed probability value of 5% for the test case termination criterion.

4.3.4. Variables and Measures

We use the following metrics to investigate our research questions:

Block coverage: This metric measures the proportion of code blocks that a test suite executes for a given AUT. A (basic) block is a sequence of code statements that always executes as a single unit [**26**].

[2]https://f-droid.org/

[3]https://play.google.com/store/apps

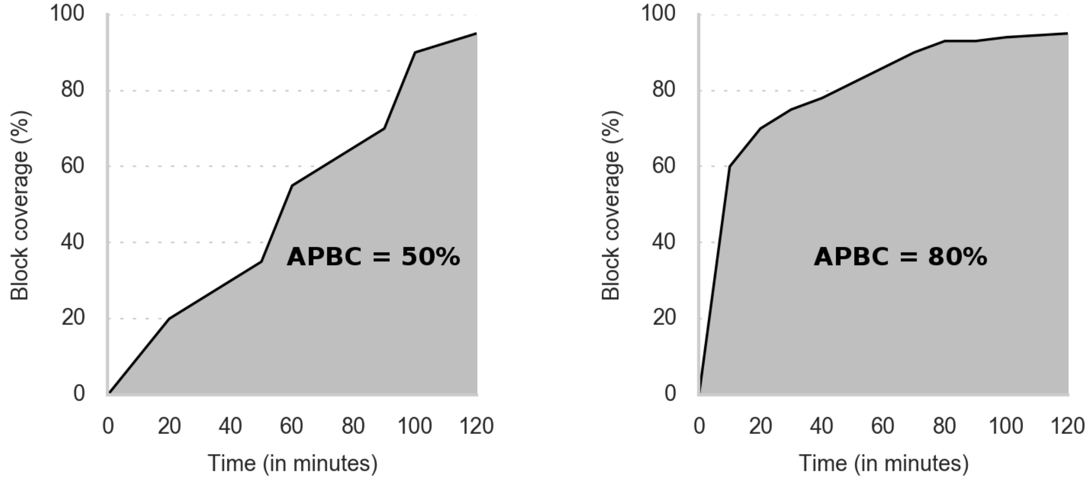


FIGURE 4.1. Examples of APBC measures

Method coverage: This metric measures the proportion of methods that a test suite executes for a given AUT.

Number of distinct events (event coverage): This metric measures the number of unique events in a test suite. It is a representation of how much of an AUT's GUI a test suite explores.

Average percentage of blocks covered (APBC): We use the APBC metric [**40**] as a measure of how quickly a test suite covers the source code of the AUT over a given time interval. The APBC metric estimates the code coverage rate of a test suite and is similar to the Average Percentage of Faults Detected (APFD) [**25**] metric often used in test case prioritization studies. The APBC of a test suite corresponds to the area under its coverage-time graph as illustrated in Figure 4.1. If t_n is the total time to generate a test suite T, t_i is some arbitrary point in time during test suite generation and $cov(t_i)$ is the block coverage at time t_i, then the APBC for the test suite is given by:

$$APBC = \frac{\sum_{i=0}^{n-1}(t_{i+1} - t_i)(cov(t_{i+1}) + cov(t_i))}{2 \times t_n \times 100} \tag{2}$$

4.3.5. Data collection

We use techniques described in Zhauniarovich et al. [**78**] to instrument the bytecode of each subject application. Bytecode instrumentation enables collection of code coverage measurements for each test suite. For each test suite, we collected code coverage measurements at time intervals that correspond to the end of each test case. Our test generation tool stores an abstract representation of the event sequences in each test suite. We analyze each test suite to collect event coverage information.

4.3.6. Statistical tests

To standardize comparisons across multiple apps, we use min-max normalization [**54**] to rescale the measurements for each application. We combine the rescaled measurements from all applications and perform Mann-Whitney U-tests [**45**] to determine whether the frequency-based test suites are significantly better than random-based test suites. We use the non-parametric Mann-Whitney U-test because it does not assume that the measurements for each dependent variable conform to a normal distribution. We consider p-values less than 0.05 to be statistically significant. A p-value less than 0.05 indicates that there is less than a 5% probability that the observed results are due to chance.

4.3.7. Results

Block coverage. Table 4.2 shows the mean block coverage across 10 test suites for each subject application and event selection strategy. The values in bold type indicate higher block coverage measurements compared to uniform random event selection. On average, the *FreqWeighted* test suites achieve up to 7% higher mean block coverage compared to the *Rand* test suites across six out of seven subject applications. The *MinFrequency* test suites achieve up to 9% higher mean block coverage compared to the *Rand* strategy across all seven subject applications.

Figure 4.2 shows the distribution of block coverage values across 10 test suites for each application and event selection strategy. The *FreqWeighted* test suites achieve higher median block coverage compared to the *Rand* test suites in six out of seven subject applications.

Application	**Block coverage (%)**		
	Rand	**FreqWeighted**	**MinFrequency**
Tomdroid	45.11	**47.52**	**47.41**
Loaned	53.53	51.83	**59.16**
Budget	66.06	**67.62**	**69.20**
A Time Tracker	70.58	**73.24**	**74.61**
Repay	47.75	**51.54**	**56.32**
Moneybalance	75.51	**82.82**	**84.69**
WhoHasMyStuff	75.59	**80.17**	**80.62**

TABLE 4.2. Mean block coverage for *Rand*, *FreqWeighted* and *MinFrequency* test suites

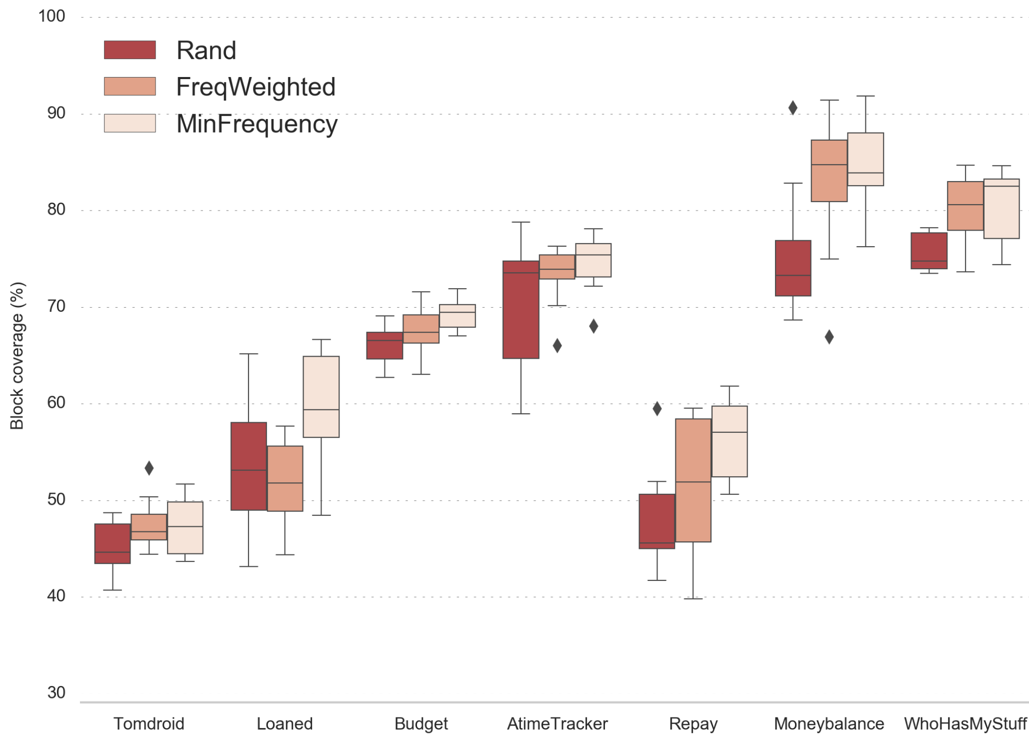


FIGURE 4.2. Boxplot of block coverage values across 10 suites for each subject application and event selection strategy

Null Hypothesis	**Alternate Hypothesis**	**p-value**
$BC(Rand) = BC(FreqWeighted)$	$BC(Rand) \neq BC(FreqWeighted)$	9.06×10^{-5}
$BC(Rand) = BC(MinFrequency)$	$BC(Rand) \neq BC(MinFrequency)$	1.10×10^{-9}
$BC(FreqWeighted) = BC(MinFrequency)$	$BC(FreqWeighted) \neq BC(MinFrequency)$	0.004

TABLE 4.3. Statistical comparison of block coverage (BC) values for *Rand*, *FreqWeighted* and *MinFrequency* test suites

The *MinFrequency* suites achieve higher median block coverage compared to the *Rand* test suites across all seven subject applications.

Table 4.3 shows the results of Mann-Whitney U-tests to compare the block coverage of the random-based and frequency-based test suites. The results show that: (i) there is a significant difference in block coverage between the *Rand* test suites and the *FreqWeighted* test suites (ii) there is a significant difference in block coverage between the *Rand* test suites and the *MinFrequency* test suites and (iii) there is a significant difference in block coverage between the *FreqWeighted* test suites and the *MinFrequency* test suites.

Method coverage. Table 4.4 shows the mean method coverage across 10 test suites for each subject application and event selection strategy. The values in bold type indicate higher method coverage measurements compared to uniform random event selection. The *FreqWeighted* test suites achieve 2-3% higher method coverage on average compared to *Rand* test suites across six out of seven subject applications. The *MinFrequency* test suites achieve 2-6% higher method coverage on average compared to *Rand* test suites across all seven subject applications.

Application	**Method coverage (%)**		
	Rand	**FreqWeighted**	**MinFrequency**
Tomdroid	47.76	**50.74**	**50.06**
Loaned	65.99	65.50	**72.01**
Budget	75.45	**77.51**	**78.70**
A Time Tracker	73.20	**76.05**	**77.33**
Repay	59.86	**62.88**	**65.82**
Moneybalance	81.90	**85.11**	**85.99**
WhoHasMyStuff	90.68	**92.48**	**92.78**

TABLE 4.4. Mean method coverage for *Rand*, *FreqWeighted* and *MinFrequency* test suites

Figure 4.3 shows the distribution of method coverage values across 10 test suites for each application and event selection strategy. The *FreqWeighted* test suites have higher median method coverage values compared to *Rand* test suites in all seven subject applications. The *MinFrequency* test suites have higher median method values compared to *Rand* test suites in all seven subject applications.

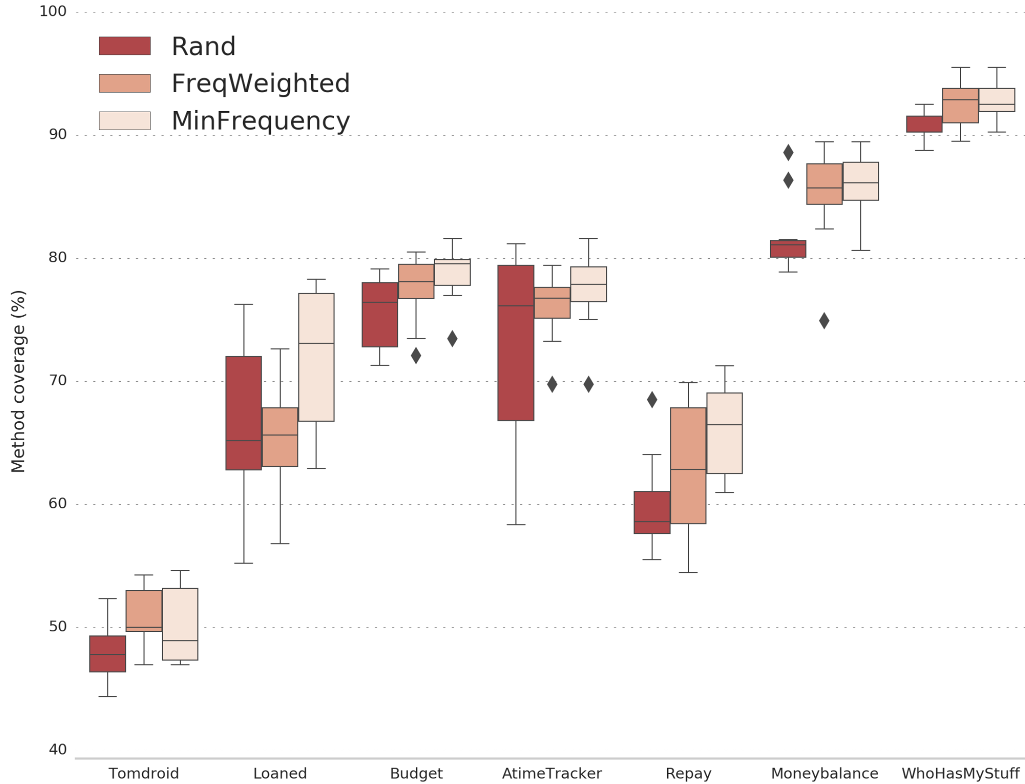


FIGURE 4.3. Distribution of method coverage values across 10 suites for each subject application and event selection strategy

Null Hypothesis	Alternate Hypothesis	p-value
$MC(Rand) = MC(FreqWeighted)$	$MC(Rand) \neq MC(FreqWeighted)$	0.0001
$MC(Rand) = MC(MinFrequency)$	$MC(Rand) \neq MC(MinFrequency)$	2.18×10^{-8}
$MC(FreqWeighted) = MC(MinFrequency)$	$MC(FreqWeighted) \neq MC(MinFrequency)$	0.03

TABLE 4.5. Statistical comparison of method coverage (MC) values for *Rand*, *FreqWeighted* and *MinFrequency* test suites

Table 4.5 shows the results of Mann-Whitney U-tests to compare the method coverage of the random-based and frequency-based test suites. The results show that: (i) there is a significant difference in method coverage between the *Rand* test suites and the *FreqWeighted* test suites (ii) there is a significant difference in method coverage between the *Rand* test suites and the *MinFrequency* test suites and (iii) there is a significant difference in method coverage between the *FreqWeighted* test suites and the *MinFrequency* test suites.

Average percentage of blocks covered (APBC). Table 4.6 shows mean APBC values for the random-based and frequency-based test suites. The APBC value for a given test suite

quantifies how quickly the test suite covers the source code of the AUT. The *FreqWeighted* test suites have a higher block coverage rate on average compared to the *Rand* test suites in six out of seven subject applications. The *MinFrequency* test suites have a higher block coverage rate on average compared to *Rand* test suites across all seven subject applications.

Application	**APBC**		
	Rand	**FreqWeighted**	**MinFrequency**
Tomdroid	0.39	**0.40**	**0.41**
Loaned	0.47	0.45	**0.51**
Budget	0.59	**0.60**	**0.62**
A Time Tracker	0.57	**0.59**	**0.63**
Repay	0.40	**0.43**	**0.47**
Moneybalance	0.67	**0.72**	**0.75**
WhoHasMyStuff	0.68	**0.72**	**0.72**

TABLE 4.6. Mean APBC values for *Rand*, *FreqWeighted* and *MinFrequency* test suites

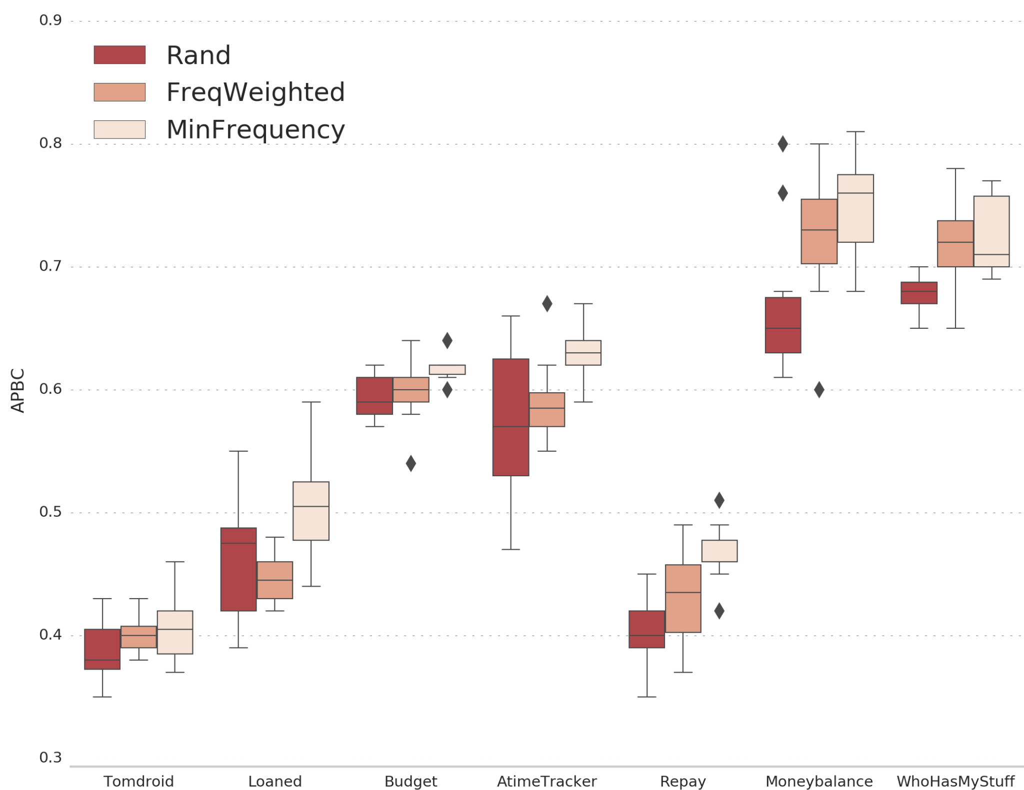


FIGURE 4.4. Boxplot of APBC values across 10 test suites for each app and event selection strategy

Figure 4.4 shows the distribution of APBC values for the random-based and frequency-based test suites. The *FreqWeighted* test suites have higher median APBC values compared to *Rand* test suites in six out of seven subject applications. The *MinFrequency* test suites have higher median APBC values compared to *Rand* test suites in all seven subject applications.

Null Hypothesis	**Alternate Hypothesis**	**p-value**
$APBC(Rand) = APBC(FreqWeighted)$	$APBC(Rand) \neq APBC(FreqWeighted)$	0.001
$APBC(Rand) = APBC(MinFrequency)$	$APBC(Rand) \neq APBC(MinFrequency)$	1.96×10^{-10}
$APBC(FreqWeighted) = APBC(MinFrequency)$	$APBC(FreqWeighted) \neq APBC(MinFrequency)$	3.45×10^{-5}

TABLE 4.7. Statistical comparison of APBC values for *Rand*, *FreqWeighted* and *MinFrequency* test suites

Table 4.7 shows the results of Mann-Whitney U-tests to compare the block coverage rates of the random-based and frequency-based test suites. The results show that: (i) there is a significant difference in block coverage rate between the *Rand* test suites and the *FreqWeighted* test suites (ii) there is a significant difference in block coverage between the *Rand* test suites and the *MinFrequency* test suites and (iii) there is a significant difference in block coverage rate between the *FreqWeighted* test suites and the *MinFrequency* test suites.

Figures 4.5-4.11 show coverage-time graphs of test suites that achieve the highest block coverage across 10 trials for each subject application and event selection strategy. The results show that the best *FreqWeighted* and *MinFrequency* test suites (in terms of block coverage) achieve similar or better code coverage rates than random-based test suites in the majority of subject applications. In some instances (e.g. Figure 4.6 and Figure 4.8), the best random-based test suites have similar or better code coverage rates than the best *FreqWeighted* and *MinFrequency* test suites.

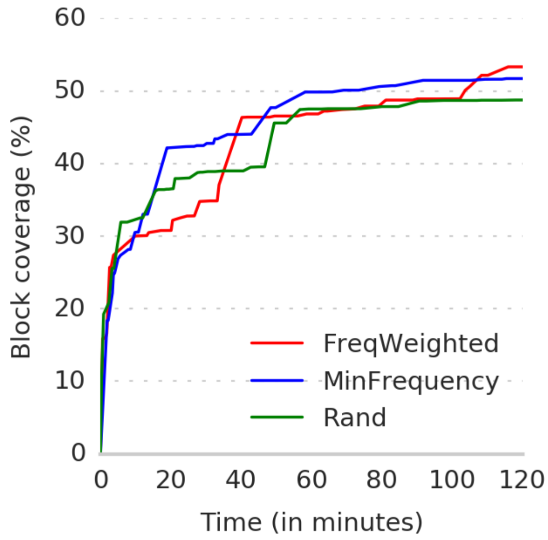


FIGURE 4.5. Coverage-time graph for *Tomdroid*

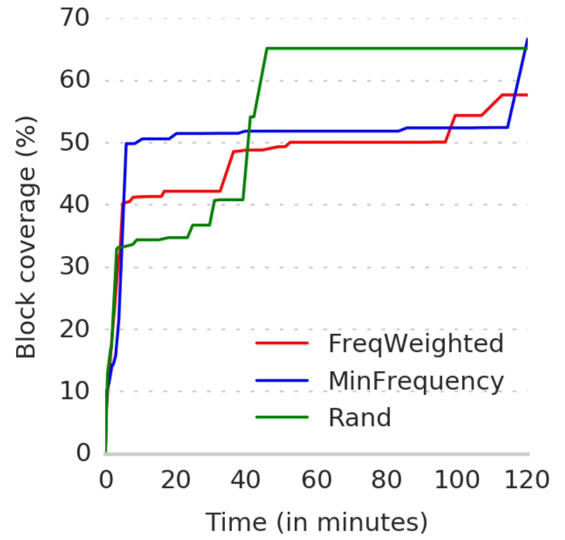


FIGURE 4.6. Coverage-time graph for *Loaned*

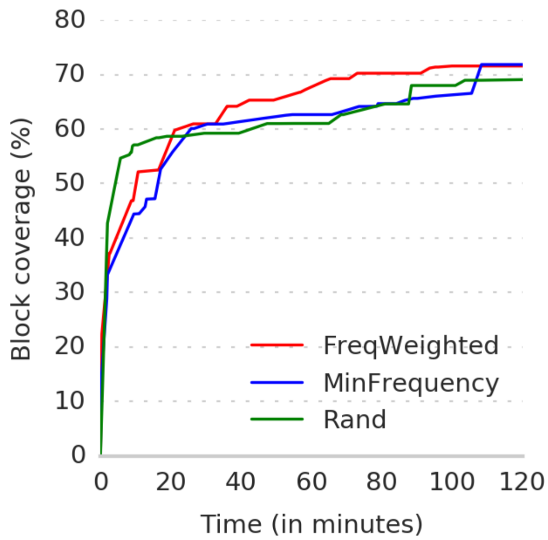


FIGURE 4.7. Coverage-time graph for *Budget*

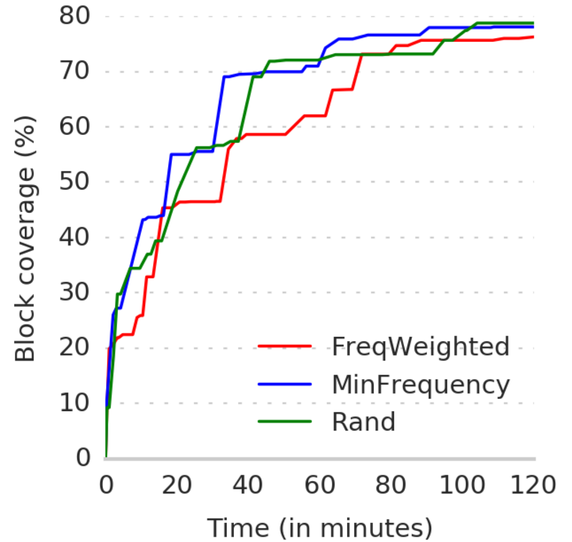


FIGURE 4.8. Coverage-time graph for *ATimeTracker*

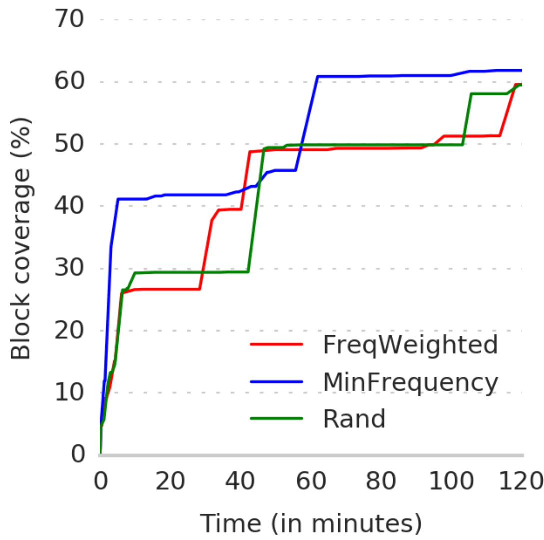


FIGURE 4.9. Coverage-time graph for *Repay*

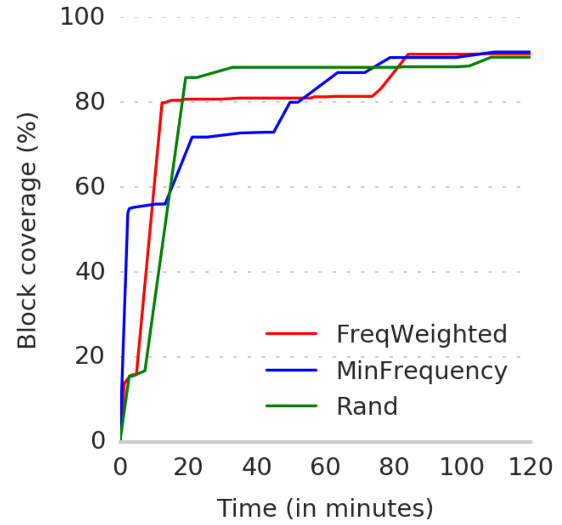


FIGURE 4.10. Coverage-time graph for *Moneybalance*

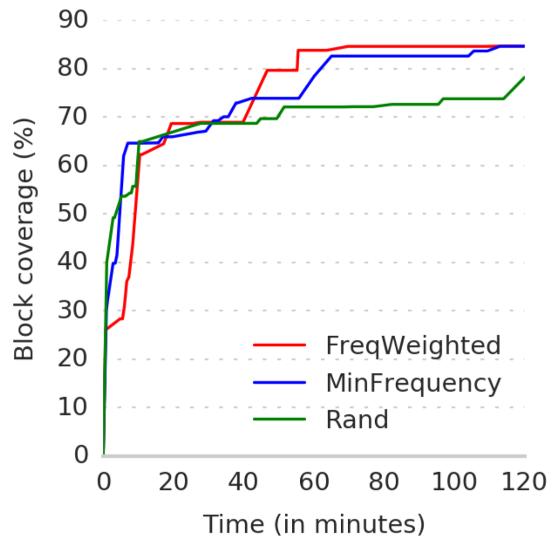


FIGURE 4.11. Coverage-time graph for *WhoHasMyStuff*

Number of distinct events (event coverage). Table 4.8 shows the average number of distinct events executed across 10 test suites for each subject application and event selection strategy. The *FreqWeighted* test suites execute 5-22 more distinct events on average compared to *Rand* test suites across all seven subject applications. The *MinFrequency* test suites execute 12-53 more distinct events on average compared to *Rand* test suites across all seven subject applications.

Application	Event coverage (%)		
	Rand	FreqWeighted	MinFrequency
Tomdroid	258	**280**	**282**
Loaned	140	**153**	**153**
Budget	243	**263**	**296**
A Time Tracker	110	**117**	**122**
Repay	115	**128**	**137**
Moneybalance	190	**200**	**224**
WhoHasMyStuff	145	**150**	**157**

TABLE 4.8. Average number of distinct events (event coverage) for *Rand*, *FreqWeighted* and *MinFrequency* test suites

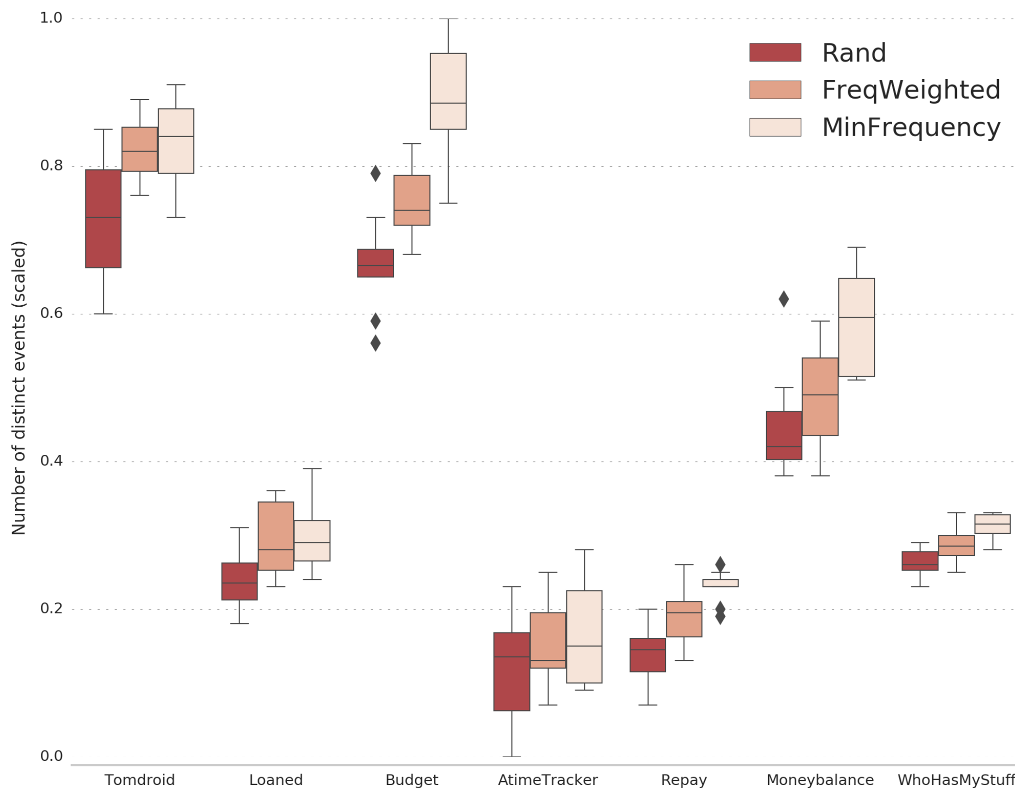


FIGURE 4.12. Distribution of event coverage values across 10 suites for each subject application and event selection strategy

Figure 4.12 shows the distribution of event coverage values for the random-based and frequency-based test suites. The *FreqWeighted* test suites have higher median event coverage values compared to *Rand* test suites in six out of seven subject applications. The *MinFrequency* test suites have higher median event coverage values compared to *Rand* test suites in all seven subject applications.

Table 4.9 shows the results of Mann-Whitney U-tests to compare the event coverage of the random-based and frequency-based test suites. The results show that: (i) there is a significant difference in event coverage between the *Rand* test suites and the *FreqWeighted* test suites (ii) there is a significant difference in event coverage between the *Rand* test suites and the *MinFrequency* test suites and (iii) there is a significant difference in event coverage between the *FreqWeighted* test suites and the *MinFrequency* test suites.

Null Hypothesis	**Alternate Hypothesis**	**p-value**
$EC(Rand) = EC(FreqWeighted)$	$EC(Rand) \neq EC(FreqWeighted)$	4.96×10^{-7}
$EC(Rand) = EC(MinFrequency)$	$EC(Rand) \neq EC(MinFrequency)$	9.05×10^{-15}
$EC(FreqWeighted) = EC(MinFrequency)$	$EC(FreqWeighted) \neq EC(MinFrequency)$	2.77×10^{-5}

TABLE 4.9. Statistical comparison of event coverage (EC) values for *Rand*, *FreqWeighted* and *MinFrequency* test suites

4.3.8. Discussion and Implications

The frequency-based event selection algorithms require a dynamically updated cache of events and the number of times they have been previously executed during test suite construction. The *FreqWeighted* and *MinFrequency* strategies compute event selection probabilities during test construction to encourage selection of events that have been previously executed fewer times relative to other events in a given GUI state. This ensures that previously unexecuted events in a given GUI state have a higher chance of selection compared to events that have already been executed at least once. This may be a factor in the significantly higher number of distinct events in the frequency-based test suites compared to the random-based test suites. Event handlers constitute a significant portion of source code for GUI-based applications. The frequency-based event selection strategies prioritize selection of previously unexecuted events and this may be a factor in the significantly higher overall code coverage and code coverage rates of the frequency-based test suites compared to the random-based test suites. Compared to uniform random event selection, the frequency-based event selection algorithms are less likely to select the same event repeatedly within a short time interval. The *FreqWeighted* event selection strategy encourages selection of previously unexecuted events but does not guarantee that such events will be selected whenever avail-

able. In one subject application (*Loaned*), the *FreqWeighted* test suites achieve higher event coverage on average compared to random-based test suites but do not show an increase in code coverage. This result indicates that increased event coverage does not always lead to increased code coverage. The *MinFrequency* strategy considers only the subset of available events that have been previously executed the least number of times in a given GUI state. Unlike the *FreqWeighted* strategy, the *MinFrequency* event selection strategy guarantees selection of previously unexecuted events whenever such events are available in a given GUI state. Compared to the *FreqWeighted* strategy, the *MinFrequency* strategy uses a more aggressive approach to minimize redundant event execution and is much less likely to select the same event disproportionately more often than other events in a given GUI state. This may be the reason why the *MinFrequency* test suites achieve the most significant improvement in overall code coverage, code coverage rate and event coverage compared to random-based test suites.

The results of our experiments show that the choice of event selection strategy has a significant impact on the cost-effectiveness of online test suite construction algorithms. The frequency-based event selection strategies outperform uniform random event selection in terms of code coverage, APBC and event coverage. It is important for an event selection algorithm to prioritize selection of previously unexecuted events and minimize repeated selection of events that do not result in additional code coverage and event coverage. Online test suite construction algorithms that use information about previously executed events to guide subsequent selection of events are likely to be more cost-effective than algorithms that select and execute events uniformly at random.

4.3.9. Threats to Validity

The principal threat to validity of this study is the generalizability of the results as we use a limited number of subject applications. The size and complexity of the AUT may affect the results obtained with our techniques. We minimized this threat by selecting apps of different sizes. The randomized nature of the event selection algorithms is also a threat to

validity. To minimize this threat, we used each event selection strategy to generate 10 test suites for each application.

4.4. Summary and Conclusions

In this chapter, we describe a frequency-based approach to online construction of Android application test suites. Our test suite construction algorithm maintains a cache of events and their prior execution frequencies during test suite construction. We develop two frequency-based event selection strategies that use frequency information about previously executed events to minimize redundant event execution and prioritize selection of new events. Both strategies dynamically alter event selection probabilities based on the prior execution frequency of events during test suite construction. We perform experiments on seven Android applications and compare our frequency-based techniques to a test suite construction algorithm that selects and executes events uniformly at random. The results show that our frequency-based approach shows significant improvements in code coverage and event coverage compared to a random-based approach. These improvements indicate that the choice of event selection strategy has a significant impact on the the effectiveness of online test generation algorithms and that it is important to use event selection strategies that minimize redundant event execution and prioritize selection of previously unexecuted events.

A significant portion of the source code in GUI-based applications defines event handlers that are executed when users perform GUI actions. The assumption behind the frequency-based algorithms in this chapter is that there is a positive correlation between the number of distinct GUI events in a test suite and code coverage. In practice, GUI-based applications (and Event Driven Systems in general) exhibit behavior and may contain failures that are triggered not just by individual events, but also by interactions among events that occur in a particular order. In the next chapter, we describe a combinatorial-based technique that considers the order in which events have been previously executed and maximizes coverage of n-way event combinations as part of the test suite construction process.

CHAPTER 5

COMBINATORIAL-BASED TEST SUITE CONSTRUCTION

Android applications are Event Driven Systems (EDSs) that take Graphical User Interface (GUI) event sequences as input and respond by changing their state. Examples of GUI events include clicking a button or entering data in a text field. EDS often include functionality that can be tested only when specific events in a sequence occur in a particular order and interactions among these events may cause a System Under Test (SUT) to enter a failure state. Combinatorial-based methods are able to systematically generate event combinations to test EDSs where the order of events is important [**36**]. These methods often require adaptation to the specific constraints imposed by GUI-based software such as mobile applications. As with other types of EDS, it may be important to test a mobile application's response to specific events executed in a particular order. Combinatorial-based testing is challenging because the number of possible event combinations in GUI-based software increases exponentially with the number of events. Combinatorial-based methods for event sequence testing manage this complexity by systematically examining combinations for only a subset of events [**30, 36, 59**]. Empirical studies in combinatorial testing show that testing interactions among a small number of inputs may be an effective way to detect software faults [**16, 37, 38, 74**].

In this chapter, we describe a combinatorial-based technique for online construction of Android application test suites. The combinatorial-based technique prioritizes execution of new events in a given GUI state, considers the order in which events have previously been executed and does not require static analysis of source code or preexisting abstract models of the Application Under Test (AUT). Our combinatorial-based test suite construction algorithm maintains a cache of executed events and the order in which they were executed relative to other events. It uses information about previously executed event combinations to greedily select and execute events that maximize coverage of n-event-tuples (i.e. n-way event combinations), where n is a specified event combination strength. This technique en-

ables automatic construction of test suites with an increased likelihood of testing behavior that occurs only when events are executed in a particular order within a single test case.

Existing techniques for automated GUI testing pay limited attention to combinatorial-based testing of mobile applications and often require static analysis of source code or construction of static behavioral models prior to test generation [**2**, **5**, **23**, **53**, **58**, **68**, **69**, **74**, **75**]. It is difficult to construct accurate models of GUI-based software and testers may not have access to the AUT's source code. Prior work in online GUI testing of Android apps uses random-based and frequency-based algorithms to generate and execute event sequences [**2**,**29**,**32**,**42**,**56**]. These algorithms often select and execute GUI events uniformly at random and have a tendency to redundantly execute events without consideration for the order in which the events have previously occurred. Since our combinatorial-based technique maximizes coverage of event combinations, it may be effective for testing Android app behavior that occurs only when events are executed in a particular order.

5.1. Combinatorial-based Test Suite Construction Algorithm

Algorithm 6 shows our online combinatorial-based algorithm that automatically constructs Android application test suites and maximizes coverage of event tuples. An n-event-tuple $(e_1, e_2, ..., e_n)$ is covered in a test suite if there exists at least one test case that contains all n events in the tuple such that $e_1 \prec e_2 \prec ... \prec e_n$[1]. For example, the test suite $T = (\langle e_1, e_2, e_3, e_4\rangle, \langle e_4, e_5, e_6, e_7\rangle)$ covers the following 3-event-tuples:
$\{(e_1, e_2, e_3), (e_1, e_3, e_4), (e_1, e_2, e_4), (e_2, e_3, e_4), (e_4, e_5, e_6), (e_4, e_6, e_7), (e_4, e_5, e_7), (e_5, e_6, e_7)\}$. The algorithm requires the following input: (i) an Android Application Package (APK) file (ii) the required event combination strength (iii) a test case termination criterion and (iv) a test suite completion criterion. The test case termination criterion may be a fixed number of events or some probabilistic criterion (as described in Chapter 3). The test suite completion criterion may be a specific number of test cases or a fixed time budget. An event combination strength of n specifies that the algorithm should maximize coverage of n-event-tuples (i.e. valid combinations of n events) during test suite construction. The algorithm maintains a

[1] $e_1 \prec e_2$ denotes that e_1 precedes e_2 in a given sequence of events

set of termination events and a set of covered event-tuples to facilitate greedy selection and execution of events that cover the largest number of uncovered tuples.

Algorithm 6: Combinatorial-based test suite construction

```
   Input  : application under test, AUT
   Input  : event combination strength, n
   Input  : test case termination criteria, t_end
   Input  : test suite completion criterion, T_comp
   Output: test suite, T
1  T ← φ                                          ▷ set of event sequences (test suite)
2  E_term ← φ                                     ▷ set of termination events to avoid
3  coveredTuples ← φ                              ▷ set of covered k-event-tuples (k ≤ n)
4  while test suite completion criteria not true do
5  |   clear application data and start AUT
6  |   t_i ← φ                                    ▷ event sequence (test case)
7  |   repeat
8  |   |   E_all ← getAvailableEvents()
9  |   |   E_all ← removeTerminationEvents(E_all, E_term)
10 |   |   C ← selectCandidates(E_all, t_i, coveredTuples, n)
11 |   |   e_sel ← breakTies(C)
12 |   |   execute and synthesize event e_sel
13 |   |   if application is closed then
14 |   |   |   E_term ← E_term ∪ {e_sel}           ▷ add to set of termination events
15 |   |   end
16 |   |   update covered event tuples in coveredTuples
17 |   |   t_i ← t_i ∪ {e_sel}
18 |   until test case termination criteria is true
19 |   T ← T ∪ t_i
20 end
```

The event sequence generation process in Algorithm 6 uses an event extraction cycle that consists of the following steps:

Step 1: Sequence Initialization. Line 6 initializes an empty test case. Before construction of each test case, line 5 of the algorithm clears any data generated by previous test cases and restarts the AUT from its initial GUI state. This ensures that all test cases start from the same GUI state and the results of one test case do not affect the behavior of subsequent test cases.

Step 2: Event identification. The *getAvailableEvents* procedure call on line 8 identifies all available events that can be executed in the current GUI state of the AUT and constructs an abstract representation of each event. In Android devices, an event associated with the "back" navigation button is always available in each GUI state. The product of this step is a set of available events E_{all} in the current GUI state.

Step 3: Candidate selection. The algorithm selects a subset of the events identified in step 2. The *removeTerminationEvents* procedure call on line 9 uses the set of termination events E_{term} defined on line 2 to remove any known termination events from the set of available events E_{all}. The *selectCandidates* procedure call on line 10 selects a set of *candidate events* C from the set of available events E_{all} such that each event $e_i \in C$ covers the highest number of uncovered event-tuples if added to the test case under construction. The *selectCandidates* procedure is defined in Algorithm 10.

Step 4: Tie breaking. The *breakTies* procedure call on line 11 handles instances where there is more than one event in the set of candidate events C. This step uses a tie-breaking strategy to choose a single event from C. In this work, we break ties at random. This step produces a single event e_{sel} to be executed on the AUT.

Step 5: Event execution. Line 12 executes the selected event e_{sel} from step 4 by interacting directly with the GUI of the application under test. Event execution often covers a set of previously uncovered event tuples and may cause the AUT to change its GUI state. Event execution may also close the AUT or explore beyond its boundaries. Lines 13-15 update the set of termination events E_{term} each time the algorithm executes a previously unknown termination event. This prevents repeated execution of termination events and encourages exploration of the AUT.

Step 6: Coverage update. After event execution, line 16 of the algorithm adds the newly covered event tuples (if any) to the set of covered event-tuples.

Step 7: Sequence update. Line 17 adds an abstract representation of the executed event to the test case under construction.

The algorithm uses multiple iterations of the event extraction cycle (steps 2-7) to

construct and execute a single test case one-event-at-a-time until it satisfies a predefined test case termination criterion. The algorithm constructs and executes multiple test cases until it meets the predefined criterion for test suite completion (e.g. a specified number of test cases or a fixed time budget).

Algorithm 7: Greedy n-way event selection

```
   Input  : set of available events in current GUI state, E_all
   Input  : test case under construction, t_i
   Input  : set of covered event tuples, coveredTuples
   Input  : event combination strength, n
   Output: set of candidate events, C
1  function selectCandidates(t_i, E_all, coveredTuples, n)
2      maxCount ← 0
3      C ← ϕ                                          ▷ set of candidate events
4      for event e_i in E_all do
5          tupleCount ← 0
6          if |t_i| < n then
7              tuples ← {t_i}
8          else
9              tuples ← generateTuples(t_i, n − 1)
10         end
11         for tuple in tuples do
12             eventTuple ← tuple ∪ {e_i}
13             if eventTuple ∉ coveredTuples then
14                 tupleCount ← tupleCount + 1
15             end
16         end
17         if tupleCount > maxCount then
18             C ← ϕ
19             C ← C ∪ {e_i}
20             maxCount ← tupleCount
21         else if tupleCount = maxCount then
22             C ← C ∪ {e_i}
23         end
24     end
25     return C
```

5.1.1. Candidate Event Selection

Algorithm 7 shows pseudocode to select a set of candidate events C from the set of available events E_{all} such that each event $e_i \in C$ covers the highest number of uncovered event tuples. The event selection algorithm requires the following input to generate the set of candidates: (i) the set of available events E_{all} in the current GUI state (ii) the test case under construction t_i (iii) the event combination strength n and (iv) the set of already covered event tuples.

If the test case under construction t_i has fewer events than the specified coverage strength n, the algorithm computes the number of new $|t_i + 1|$-event tuples that will be covered by each available event $e_i \in E_{all}$. If $|t_i| \geq n$, the algorithm generates all $(n-1)$-event-tuples in the test case t_i (line 9) and computes the number of n-event-tuples that each available event will cover if added to the test case. This incremental process ensures that the algorithm always prioritizes coverage of new event combinations even when the test case t_i does not yet contain enough events to satisfy the specified event combination strength n. For example, if the specified event combination strength n is 4, and the test case at the time of event selection is $t_i = (e_1)$, the algorithm will compute the number of new event pairs that will be covered for each $e_i \in E_{all}$. Similarly, when $t_i = (e_1, e_2)$ with $n = 4$, the algorithm will count the number of new event triples, and so on until $|t_i| \geq n$. When $|t_i| \geq n$, the algorithm computes the number of new n-event-tuples that will be covered if a given e_i in E_{all} is added to the test case. The output of Algorithm 7 is a set of events that cover the highest number of uncovered event tuples.

5.2. Experiments

5.2.1. Research Questions

We describe experiments to address the following research questions:

- **RQ1:** Does 2-way and 3-way combinatorial-based test suites increase code coverage and rate of code coverage compared to random-based test suites?

- **RQ2:** Does 2-way and 3-way combinatorial-based test suites increase event coverage compared to random-based test suites?
- **RQ3:** How does an increase in event combination strength from 2-way to 3-way affect code coverage and rate of code coverage?
- **RQ4:** How does an increase in event combination strength from 2-way to 3-way affect event coverage?

5.2.2. Subject Applications

App Name	**Lines**	**Methods**	**Classes**	**Activities**
Tomdroid v0.7.2	5,736	496	131	8
Loaned v1.0.2	2,837	258	70	4
Budget v4.0	3,159	367	67	8
A Time Tracker v0.23	1,980	130	22	5
Repay v1.6	2,059	204	48	6
Moneybalance v1.0	1,460	163	37	5
WhoHasMyStuff v1.0.25	1,026	90	24	2

TABLE 5.1. Characteristics of selected Android apps

We evaluate the combinatorial-based technique on seven Android apps retrieved from the F-droid Android app repository[2]. Table 5.1 shows the characteristics of the selected apps. The apps range from 1,026 to 5,736 source lines of code (SLOC), 90 to 496 methods, 24 to 131 classes and 2 to 8 activities. Since our implementation relies on Android GUI testing libraries, we select apps with GUIs that predominantly use the standard widgets provided by the Android framework. We limit our selection of apps to those that allow automatic bytecode instrumentation [**78**] without direct modification of source code.

5.2.3. Experimental Setup

Our experiments examine the following techniques:

Random-based (Rand). This technique selects and executes events uniformly at random.

[2]http://f-droid.org

2-way combinatorial-based. This technique uses the combinatorial-based algorithm to greedily select and execute events that maximize coverage of event pairs (i.e. 2-way event combinations).

3-way combinatorial-based. This technique uses the combinatorial-based algorithm to greedily select and execute events that maximize coverage of 3-event-tuples (i.e. 3-way event combinations).

We perform our experiments on Android 4.4.4 emulator instances with 4 processors and 2GB RAM. We generate 10 test suites using each criteria (random-based, 2-way combinatorial-based and 3-way combinatorial-based) for each subject application. We use a fixed time budget of two hours (120 minutes) to generate each test suite. We set a two-second delay between execution of consecutive events in each test case to give the AUT time to respond to each event. We set the probability of test case termination to 5%. Most open source Android applications either have no test suites or have test suites that only cover up to 40% of the application's source code [**35**]. We chose the 5% probability value because it enables the random-based (i.e. the baseline) algorithm to consistently generate test suites that achieve an average of at least 40% code coverage within the two-hour time budget and in all our subject applications.

5.2.4. Variables and measures

We use the following metrics to investigate our research questions:

Block coverage: This metric measures the proportion of code blocks that a test suite executes for a given AUT. A (basic) block is a sequence of code statements that always executes as a single unit [**26**].

Method coverage: This metric measures the proportion of methods that a test suite executes for a given AUT.

Number of distinct events (event coverage): This metric measures the number of unique events in a test suite. It is a representation of how much of an AUT's GUI a test suite explores.

Average percentage of blocks covered (APBC): We use the APBC metric [**40**] as a measure of how quickly a test suite covers the source code of the AUT over a given time interval. The APBC metric estimates the code coverage rate of a test suite and is similar to the Average Percentage of Faults Detected (APFD) [**25**] metric often used in test case prioritization studies. The APBC of a test suite corresponds to the area under its coverage-time graph. If t_n is the total time to generate/execute a test suite T, t_i is some arbitrary point in time during test suite generation and $cov(t_i)$ is the block coverage at time t_i, then the APBC for the test suite is given by:

$$APBC = \frac{\sum_{i=0}^{n-1}(t_{i+1} - t_i)(cov(t_{i+1}) + cov(t_i))}{2 \times t_n \times 100} \tag{3}$$

5.2.5. Implementation

We extend our automated GUI testing tool, Autodroid, to include a *combinatorics engine* that computes and tracks event combinations during test suite construction. Autodroid's combinatorics engine and event selector use the combinatorial information to greedily select events that maximize coverage of uncovered event combinations.

5.2.6. Data collection

We use techniques described in Zhauniarovich et al. [**78**] to instrument the bytecode of each subject application. Bytecode instrumentation enables collection of code coverage measurements for each test suite. For each test suite, we collected code coverage measurements at time intervals that correspond to the end of each test case. Our test generation tool stores an abstract representation of the event sequences in each test suite. We analyze each test suite to collect event coverage information.

5.2.7. Statistical tests

To standardize comparisons across multiple apps, we use min-max normalization [**54**] to rescale the measurements for each application. We combine the rescaled measurements from all applications and perform Mann-Whitney U-tests [**45**] to determine whether the

combinatorial-based test suites are significantly better than random-based test suites. We use the non-parametric Mann-Whitney U-test because it does not assume that the measurements for each dependent variable conform to a normal distribution. We consider p-values less than 0.05 to be statistically significant. A p-value less than 0.05 indicates that there is less than a 5% probability that the observed results are due to chance.

5.2.8. Results

Block coverage. Table 5.2 shows the mean block coverage values of the random-based and combinatorial-based test suites for each application. The values in bold type indicate higher block coverage measurements compared to the random-based test suites. The 2-way combinatorial-based test suites achieve 1.5% - 9.9% higher mean block coverage compared to random-based test suites across all seven subject applications. The 3-way combinatorial-based test suites achieve 0.2% - 6.4% higher mean block coverage compared to random-based test suites across all seven subject applications. The 3-way combinatorial-based test suites achieve higher mean block coverage than the 2-way combinatorial-based test suites in two out of seven applications.

Figure 5.1 shows the distribution of block coverage values for the random-based and combinatorial-based test suites. The 2-way and 3-way combinatorial-based test suites achieve higher median block coverage than random-based test suites in all seven subject applications. In three out of seven subject applications, 3-way combinatorial-based test suites achieve higher median block coverage than 2-way combinatorial-based test suites.

Table 5.3 shows the results of Mann-Whitney U-tests to compare the block coverage of the random-based, 2-way combinatorial-based and 3-way combinatorial-based test suites. The results show that: (i) there is a significant difference in block coverage between the random-based test suites and the 2-way combinatorial-based test suites (ii) there is a significant difference in block coverage between the random-based test suites and the 3-way combinatorial-based test suites and (iii) there is no significant difference in block coverage between the 2-way combinatorial-based test suites and the 3-way combinatorial-based test suites.

Application	Block coverage (%)		
	Rand	2-way	3-way
Tomdroid	45.11	**46.57**	**45.28**
Loaned	53.53	**56.90**	**59.40**
Budget	66.06	**69.77**	**69.66**
A Time Tracker	70.58	**75.19**	**73.40**
Repay	47.75	**57.10**	**54.19**
Moneybalance	75.51	**85.39**	**81.74**
WhoHasMyStuff	75.59	**81.15**	**81.17**

TABLE 5.2. Mean block coverage of random-based and combinatorial-based test suites

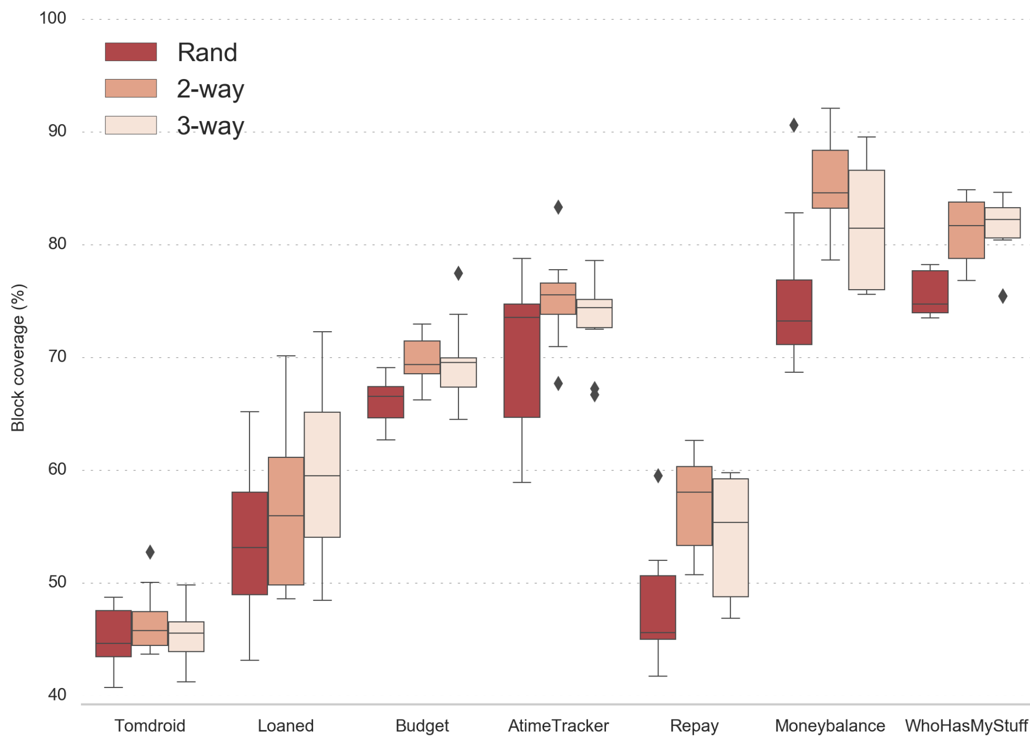


FIGURE 5.1. Boxplot of block coverage values across 10 suites for each app and technique

Null Hypothesis	Alternate Hypothesis	p-value
$BC(Rand) = BC(2way)$	$BC(Rand) \neq BC(2way)$	6.5×10^{-10}
$BC(Rand) = BC(3way)$	$BC(Rand) \neq BC(3way)$	3.8×10^{-7}
$BC(2way) = BC(3way)$	$BC(2way) \neq BC(3way)$	0.22

TABLE 5.3. Statistical comparison of block coverage (BC) values for random-based, 2-way combinatorial-based and 3-way combinatorial-based test suites

Method coverage. Table 5.4 shows the mean method coverage values of the random-based and combinatorial-based test suites. The 2-way and 3-way combinatorial-based test suites achieve higher mean method coverage compared to random-based test suites across all seven subject applications. The 3-way combinatorial-based test suites achieve higher mean method coverage than the 2-way combinatorial-based test suites in two out of seven subject applications.

Application	**Method coverage (%)**		
	Rand	**2-way**	**3-way**
Tomdroid	47.76	**50.12**	**48.59**
Loaned	65.99	**69.68**	**71.52**
Budget	75.45	**79.32**	**78.48**
A Time Tracker	73.20	**78.16**	**76.93**
Repay	59.86	**67.16**	**64.31**
Moneybalance	81.90	**86.78**	**85.51**
WhoHasMyStuff	90.68	**91.73**	**92.86**

TABLE 5.4. Mean method coverage of random-based and combinatorial-based test suites

Figure 2 shows the distribution of method coverage values for the random-based and combinatorial-based test suites. The 2-way and 3-way combinatorial-based test suites achieve higher median method coverage compared to random-based test suites across all seven subject applications. In two out of seven subject applications, 3-way combinatorial-based test suites achieve higher median method coverage than 2-way combinatorial-based test suites.

Null Hypothesis	**Alternate Hypothesis**	**p-value**
$MC(Rand) = MC(2way)$	$MC(Rand) \neq MC(2way)$	2.1×10^{-8}
$MC(Rand) = MC(3way)$	$MC(Rand) \neq MC(3way)$	1.5×10^{-6}
$MC(2way) = MC(3way)$	$MC(2way) \neq MC(3way)$	0.2

TABLE 5.5. Statistical comparison of method coverage (MC) values for random-based, 2-way combinatorial-based and 3-way combinatorial-based test suites

Table 5.5 shows the results of Mann-Whitney U-tests to compare the method coverage of the random-based, 2-way combinatorial-based and 3-way combinatorial-based test suites.

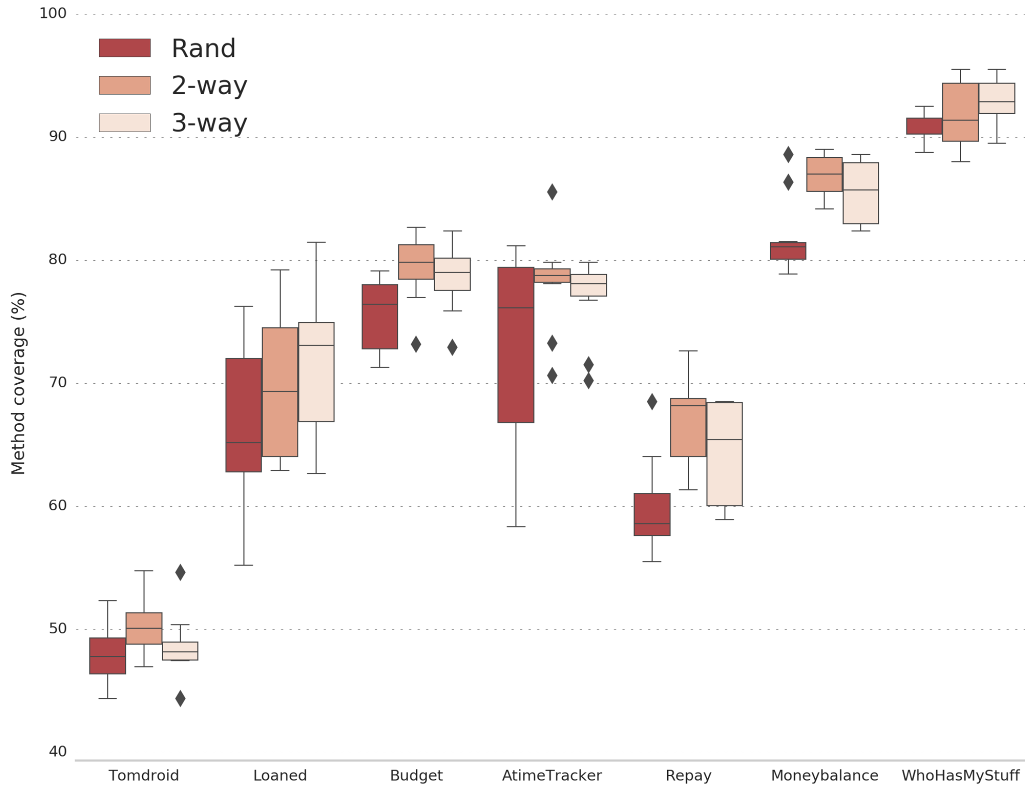


FIGURE 5.2. Boxplot of method coverage values across 11 ten suites for each app and technique

The results show that: (i) there is a significant difference in method coverage between the random-based test suites and the 2-way combinatorial-based test suites (ii) there is a significant difference in method coverage between the random-based test suites and the 3-way combinatorial-based test suites and (iii) there is NO significant difference in method coverage between the 2-way combinatorial-based test suites and the 3-way combinatorial-based test suites.

Average percentage of blocks covered (APBC). Table 5.6 shows mean APBC values for the random-based and combinatorial-based test suites. The APBC value for a given test suite quantifies how quickly the test suite covers the source code of the AUT. In all seven subject applications, the 2-way and 3-way combinatorial-based test suites achieve an equal or higher mean block coverage rate compared to random-based test suites. In three out of seven subject applications, the 3-way combinatorial-based test suites achieve an equal or higher mean block coverage rate compared to 2-way combinatorial-based test suites.

Application	APBC		
	Rand	**2-way**	**3-way**
Tomdroid	0.39	0.39	0.39
Loaned	0.47	0.47	**0.49**
Budget	0.59	**0.62**	**0.61**
A Time Tracker	0.57	**0.62**	**0.62**
Repay	0.40	**0.49**	**0.45**
Moneybalance	0.67	**0.75**	**0.71**
WhoHasMyStuff	0.68	**0.74**	**0.73**

TABLE 5.6. Mean APBC values for the random-based and combinatorial-based test suites

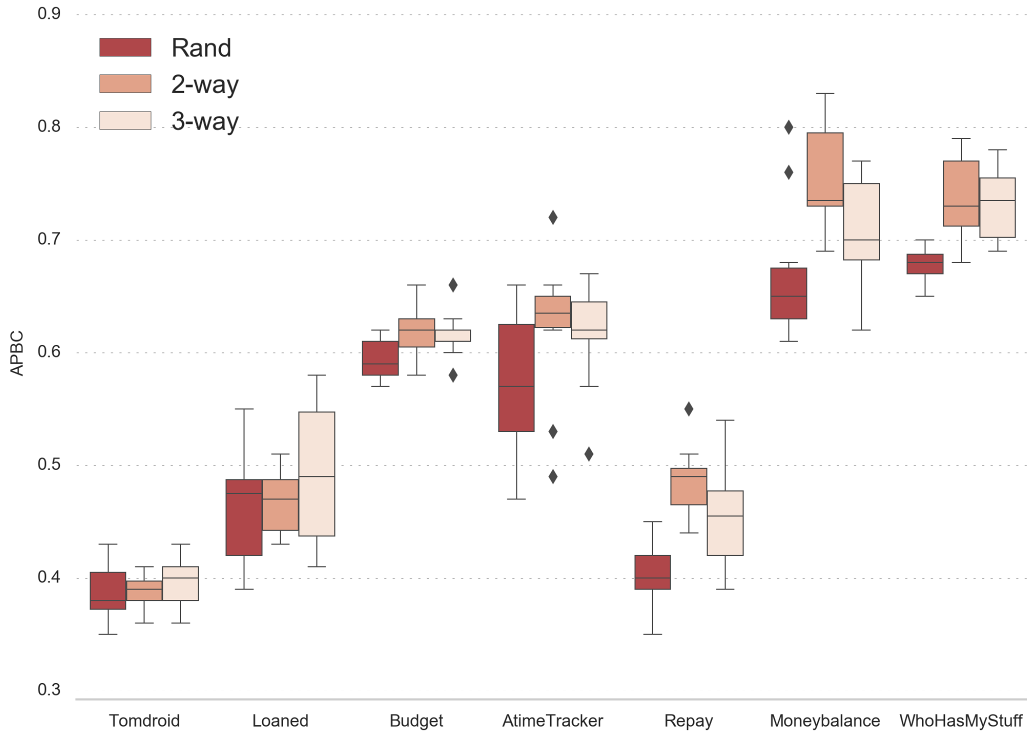


FIGURE 5.3. Boxplot of APBC values across 10 test suites for each app and technique

Figure 5.3 shows the distribution of APBC values for the random-based and combinatorial-based test suites. The 2-way combinatorial-based test suites achieve higher median APBC values than random-based test suites in six out of seven subject applications. The 3-way combinatorial-based test suites achieve higher median APBC values than random-based test suites in all seven subject applications. In three out of seven subject applications,

3-way combinatorial-based test suites achieve higher median block coverage than 2-way combinatorial-based test suites.

Null Hypothesis	**Alternate Hypothesis**	**p-value**
$APBC(Rand) = APBC(2way)$	$APBC(Rand) \neq APBC(2way)$	1.4×10^{-8}
$APBC(Rand) = APBC(3way)$	$APBC(Rand) \neq APBC(3way)$	1.6×10^{-6}
$APBC(2way) = APBC(3way)$	$APBC(2way) \neq APBC(3way)$	0.42

Table 5.7. Statistical comparison of APBC values for random-based, 2-way combinatorial-based and 3-way combinatorial-based test suites

Table 5.7 shows results of Mann-Whitney U-tests to compare the APBC values of the random-based, 2-way combinatorial-based and 3-way combinatorial-based test suites. The results show that: (i) there is a significant difference in block coverage rate between the random-based test suites and the 2-way combinatorial-based test suites (ii) there is a significant difference in block coverage rate between the random-based test suites and the 3-way combinatorial-based test suites and (iii) there is no significant difference in block coverage rate between the 2-way combinatorial-based test suites and 3-way combinatorial-based test suites.

Figures 5.4-5.10 show coverage-time graphs of the best test suites (in terms of code coverage) for each subject application and technique. The 2-way and 3-way combinatorial-based test suites achieve similar or better code coverage rates compared to random-based test suites for the majority of subject applications. The 2-way and 3-way combinatorial-based test suites that achieve the highest code coverage for Moneybalance (Figure 5.9) and Loaned (Figure 5.5) take more time to achieve similar levels of coverage as the best random-based test suites.

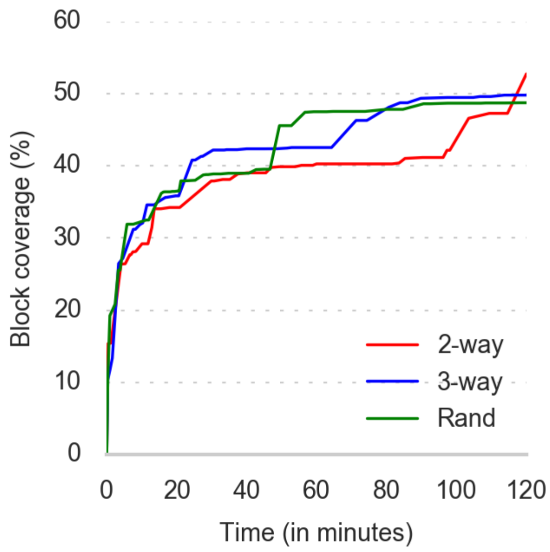


FIGURE 5.4. Coverage-time graph for *Tomdroid*

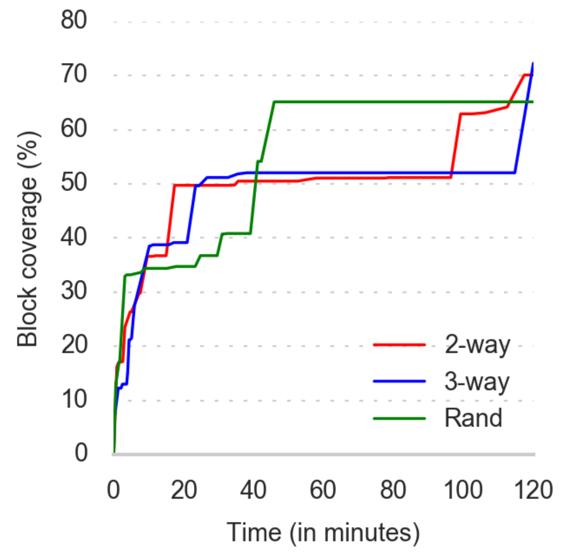


FIGURE 5.5. Coverage-time graph for *Loaned*

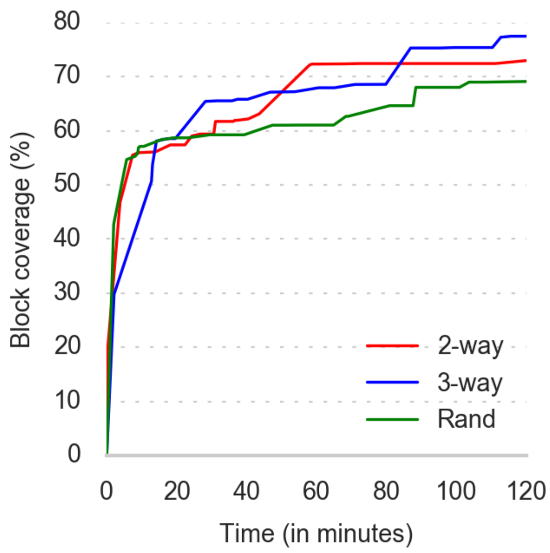


FIGURE 5.6. Coverage-time graph for *Budget*

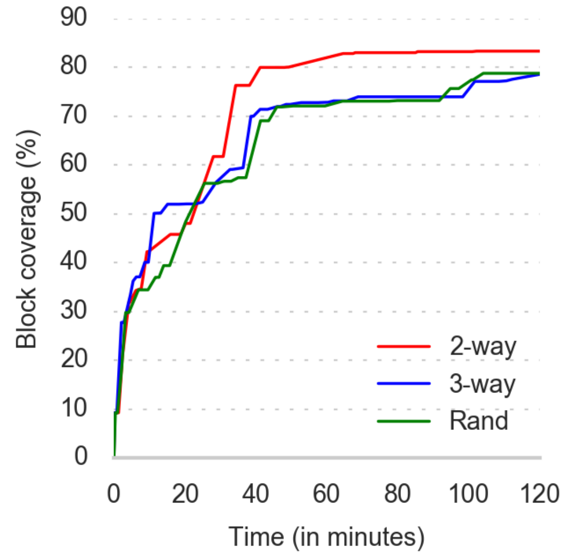


FIGURE 5.7. Coverage-time graph for *ATimeTracker*

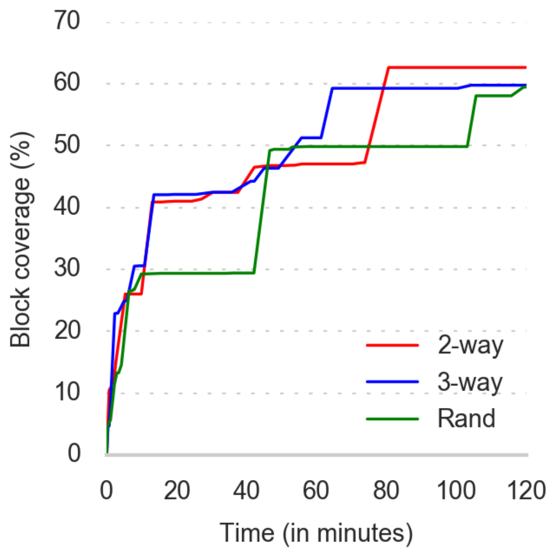


FIGURE 5.8. Coverage-time graph for *Repay*

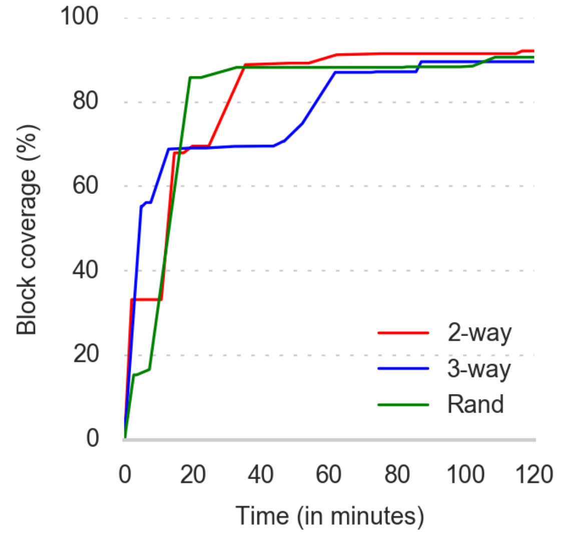


FIGURE 5.9. Coverage-time graph for *Moneybalance*

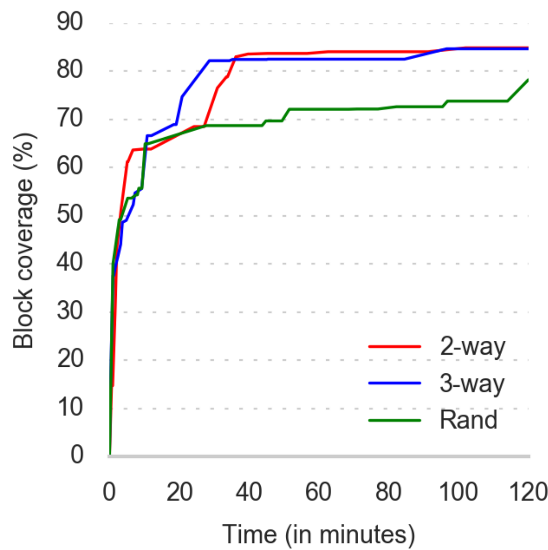


FIGURE 5.10. Coverage-time graph for *WhoHasMyStuff*

Number of distinct events (event coverage). Table 5.8 shows the average number of distinct events executed across 10 test suites for each subject application and technique. In all seven subject applications, the 2-way combinatorial-based test suites have an equal or higher number of distinct events compared to the random-based test suites. In all seven subject applications, the 3-way combinatorial-based test suites have a higher average number

of distinct events compared to the random-based test suites. In four out of seven subject applications, the 3-way combinatorial-based test suites have an equal or higher average number of distinct events compared to the 2-way combinatorial-based test suites.

Application	**Number of distinct events**		
	Rand	**2-way**	**3-way**
Tomdroid	258	**279**	**279**
Loaned	140	**155**	**155**
Budget	243	**264**	**260**
A Time Tracker	110	**121**	**120**
Repay	115	**148**	**146**
Moneybalance	190	**207**	**207**
WhoHasMyStuff	145	145	**156**

TABLE 5.8. Average number of distinct events covered (rounded to whole numbers) across 10 test suites for each subject application and technique

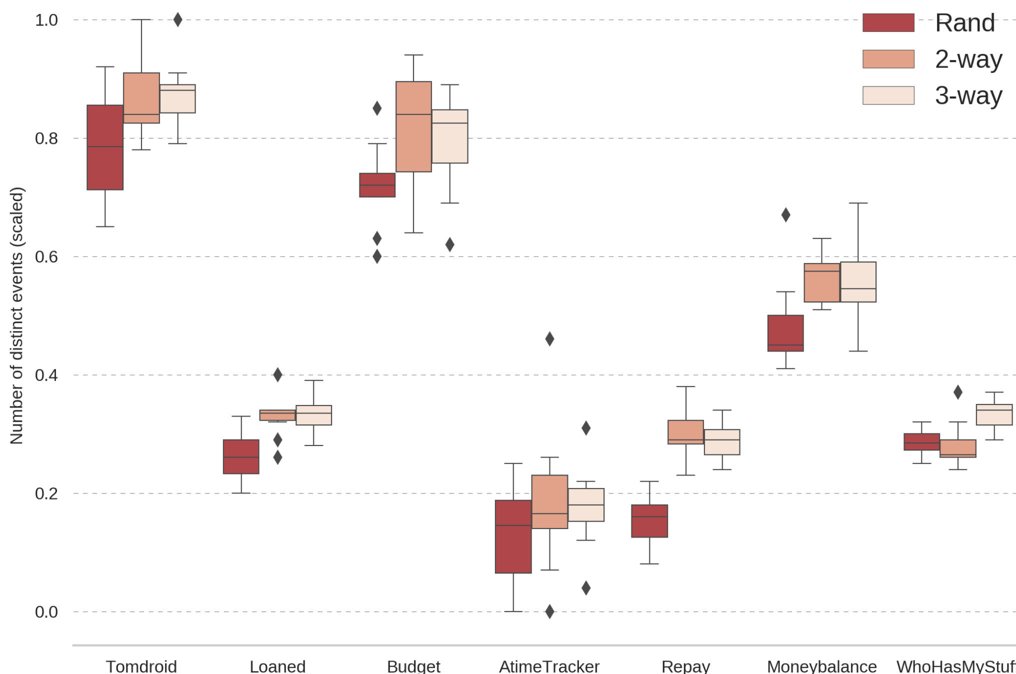


FIGURE 5.11. Boxplot of number of distinct events across 10 test suites for each app and technique

Figure 5.11 shows the distribution of event coverage values for the random-based and combinatorial-based test suites. The 2-way combinatorial-based test suites have a higher median number of distinct events compared to the random-based test suites in six out of

seven subject applications. In all seven subject applications, the 3-way combinatorial-based test suites have a higher median number of distinct events compared to the random-based test suites. The 3-way combinatorial-based test suites have an equal or higher median number of distinct events than the 2-way combinatorial-based test suites in four out of seven subject applications.

Null Hypothesis	**Alternate Hypothesis**	**p-value**
$EC(Rand) = EC(2way)$	$EC(Rand) \neq EC(2way)$	6.2×10^{-8}
$EC(Rand) = EC(3way)$	$EC(Rand) \neq EC(3way)$	9.7×10^{-12}
$EC(2way) = EC(3way)$	$EC(2way) \neq EC(3way)$	0.23

TABLE 5.9. Statistical comparison of event coverage (EC) for random-based, 2-way combinatorial-based and 3-way combinatorial-based test suites

Table 5.9 shows results of Mann-Whitney U-tests to compare the number of distinct events in the random-based, 2-way combinatorial-based and 3-way combinatorial-based test suites. The results show that: (i) there is a significant difference in event coverage between random-based test suites and 2-way combinatorial-based test suites (ii) there is a significant difference in event coverage between random-based test suites and 3-way combinatorial-based test suites and (iii) there is no significant difference in event coverage between 2-way combinatorial-based test suites and 3-way combinatorial-based test suites.

5.2.9. Discussion and Implications

Within the specified two-hour time budget for each test suite, the 2-way and 3-way combinatorial-based test suites achieve significantly higher code coverage compared to random-based test suites. The 2-way combinatorial-based test suites show the most significant improvement in code coverage compared to random-based test suites. There is no statistically significant difference in code coverage between the 2-way and 3-way combinatorial-based test suites. This result suggests that given a fixed time budget, an increase in event combination strength for combinatorial-based test suites does not necessarily increase overall code coverage. In two out of seven subject applications (*Loaned* and *WhoHasMyStuff*), 3-way combinatorial-based test suites achieve higher mean and median block/method coverage compared to 2-way combinatorial-based test suites. *WhoHasMyStuff* and *Loaned* have

two and four activities respectively and the other five subject applications in our experiments range from five to eight activities. This observation suggests that compared to 2-way combinatorial-based test suites, higher event combination strengths may be more effective (in terms of code coverage) for simple Android applications that have few activities.

The 2-way and 3-way combinatorial-based test suites show a significant improvement in event coverage compared to random-based test suites. The random-based algorithm is likely to repeatedly select events that do not provide any added benefits in terms of code coverage and event coverage. During test suite construction, events that have been selected least frequently (or not at all) are most likely to cover the highest number of new event tuples. Our combinatorial-based technique prioritizes selection and execution of new or infrequently selected events as a side effect of its maximization of event tuple coverage. This may be a factor in the improved event coverage of the combinatorial-based test suites compared to random-based test suites. There is no statistically significant difference between the event coverage of the 2-way and 3-way combinatorial-based test suites.

In two of the seven subject applications (*Loaned* and *WhoHasMyStuff*), the 3-way combinatorial-based test suites achieve faster code coverage than the 2-way combinatorial-based test suites. These subject applications are the only ones that have fewer than five activities. This suggests that given a fixed time budget, an increase in event combination strength may lead to increased code coverage rates in simple Android applications with a small number of activities.

Combinatorial-based techniques may test interactions between events that occur in a particular order. Our results demonstrate the cost-effectiveness of combinatorial-based techniques for automatic construction of Android application test suites. With a limited time budget, our combinatorial-based technique produces test suites with equal or higher code coverage, code coverage rates and event coverage compared to a random-based technique. Users of automated GUI testing tools for Android apps need to consider the complexity of of the application under test, particularly the number of lines of code, number of activities and amount of interaction between widgets before choosing an event combination strength for

combinatorial-based event sequence testing. The computational cost of our combinatorial-based algorithm increases in direct proportion to the specified event combination strength. The effectiveness of test suites produced with higher event combination strengths ($n > 2$) may depend on the available time budget and the complexity of the application under test.

5.2.10. Threats to Validity

The randomized nature of our combinatorial-based algorithm is the primary threat to the validity of this study. To minimize this threat, we experiment with 10 test suites for each application and test generation criterion. Another threat to the validity of this study is the generalizability of the results as we evaluate our combinatorial-based technique on a limited number of Android applications. The effectiveness of our approach may depend on the size and complexity of the application under test. To minimize this threat, we experiment with seven Android applications of various sizes.

We perform experiments with a fixed time budget of two hours for each test suite. A smaller or larger time budget may produce results different from those reported in this study. The results of our experiments with a two hour time budget provide useful insight into the cost-effectiveness of our combinatorial-based technique compared to a random-based technique. We use a greedy algorithm to generate event sequences that maximize coverage of event tuples. Other techniques (e.g. search-based) may yield different results.

5.3. Summary and Conclusions

This chapter presents a combinatorial-based technique to automatically construct Android application test suites and maximize coverage of n-way event combinations, where n is a predefined event combination strength. The objective is to minimize redundant execution of events and increase the likelihood of testing behavior that occurs only when events are executed in a particular order. Our online technique does not require source code analysis or static abstract models of the AUT's behavior. We evaluated our combinatorial-based technique on seven Android applications and compared it to a random-based technique. The results of our experiments show that 2-way and 3-way combinatorial-based test suites

are more effective than random-based test suites in terms of code coverage, code coverage rate and event coverage despite the additional overhead of computing event combinations. There is no statistically significant difference between 2-way and 3-way combinatorial-based test suites. Higher event combination strengths may require additional time to produce test suites that are more effective than those produced with lower event combination strengths. This is because the computational cost of our combinatorial-based algorithm increases in direct proportion to the specified event combination strength.

In Chapter 6, we describe a framework and pairwise event selection technique that enables automatic construction of Android application test suites with systematically interleaved GUI events and context events (e.g. changes in network connectivity, screen orientation and battery levels).

CHAPTER 6

TESTING CONTEXT-SENSITIVE BEHAVIOR IN ANDROID APPLICATIONS

Mobile applications are Event Driven Systems (EDSs) that pose unique testing challenges. Users typically interact with mobile applications via a Graphical User Interface (GUI). These applications take GUI event sequences as input and respond by changing their state. GUI-based software is difficult to test due to the prohibitively large number of possible GUI event sequences in the input space [**50**]. Mobile applications further complicate the testing process with their ability to respond to context events (e.g. changes in network connectivity, battery levels, location, etc.). These context events often modify one or more context variables (e.g. screen orientation, connectivity status, etc.) that define the *operating context* of a mobile application and may affect its behavior. A mobile application may react directly to context changes or respond differently to identical GUI events executed in multiple contexts. Some features of a mobile application may be accessible only in specific contextual conditions (e.g. availability of an internet connection) and faults may occur only in specific contexts or as a result of interactions between context variables. GUI states and lines of code that drive context-sensitive functionality may be unreachable with test suites that do not manipulate the operating context of the AUT. Mobile applications may respond to context changes and are often expected to function reliably in different operating contexts. It is important to develop cost-effective testing techniques that consider context changes in addition to GUI events.

There are several tools and techniques for automated GUI testing of mobile applications [**2–5**, **42**, **46**, **53**, **56**, **78**]. The majority of these tools do not consider the operating context of the AUT during test generation and execution. They execute GUI event sequences in a single predefined context and may test only the subset of an AUT's functionality that is available in the predefined context. Some prior research describes techniques to execute preexisting test suites in multiple contexts and techniques to insert context events into preexisting test suites [**1**, **44**, **64**]. These techniques may produce infeasible GUI event sequences

since context changes often alter the behavior of the AUT.

In this chapter, we develop a framework that allows testers to use different criteria to automatically generate Android application test suites that include context events and GUI events. The framework considers the operating context of an AUT to be a combination of values for a predefined set of context variables and enables testers to instantiate different test generation techniques that assess the behavior of Android applications under changing contexts. It enables testers to instantiate different test generation techniques that assess the behavior of Android applications under changing contexts. As part of our framework, we develop a pairwise event selection technique that systematically executes GUI events in different contexts, tests potential interactions between context variables and automatically regulates the frequency of context changes in relation to GUI events. The test generation techniques explored in this chapter combine context manipulation and test generation into a single process. Techniques in prior work treat context manipulation and test generation as separate activities. We hypothesize that systematic execution of GUI events in multiple contexts is a cost-effective way to manage the large input space and improve the quality of test suites for context-sensitive Android applications.

6.1. Context Modelling

Our framework requires a combinatorial context model that specifies a set of context variables and values for use during test generation. Testers may specify combinatorial context models in line with specific requirements of the AUT.

WiFi	**Battery**	**AC Power**	**Screen Orientation**
Connected	Ok	Connected	Portrait
Disconnected	Low	Disconnected	Landscape

Table 6.1. Combinatorial testing model with four context variables and two values for each variable

Table 6.1 shows a combinatorial context model with four context variables (WiFi, battery, AC power and screen orientation). Each context variable has two possible values. Hence, there are 2^4 possible value combinations. The number of context variables in the

combinatorial context model may be expanded to include other variables (e.g. GPS, bluetooth, etc.) in line with requirements of the AUT. A combinatorial context model may also define constraints between context variables. An exhaustive combination of k context variables, each with v possible values, results in v^k possible value combinations. A combinatorial context model with 10 context variables and three possible values for each variable implies that the AUT may be exhaustively tested in 3^{10} possible contexts. Exhaustive combination of values for context variables quickly becomes cost prohibitive since the number of possible combinations increases exponentially with the number of context variables. To manage this combinatorial explosion, given a combinatorial context model with k context variables, a tester may model the possible operating contexts of an AUT as a t-way covering array. For a combinatorial model with k variables and v possible values for each variable, a *t-way covering array* $CA(N; t; k; v)$ has N rows and k columns such that each t-tuple occurs at least once within the rows, where t is the strength of interaction coverage [**13**].

ID	**WiFi**	**Battery**	**AC Power**	**Screen Orientation**
c_1	Connected	Low	Disconnected	Landscape
c_2	Connected	OK	Connected	Portrait
c_3	Disconnected	Low	Connected	Landscape
c_4	Disconnected	OK	Disconnected	Portrait
c_5	Disconnected	Low	Disconnected	Portrait
c_6	Connected	OK	Disconnected	Landscape

TABLE 6.2. A 2-way covering array that defines six contexts

Table 6.2 shows a 2-way covering array for the combinatorial context model in Table 6.1. Each row of the covering array c_i represents a single operating context that will be used to assess the functionality of the AUT. Our framework generates context covering arrays from combinatorial context models to enable testing of potential interactions between context variables and reduce the combinatorial explosion when the number of context variables increases.

6.2. Definitions

The framework relies on a set of abstractions to automatically construct test suites for context-sensitive Android applications. We define *context*, *context-GUI event pair coverage*

and *context-state pair coverage* as follows.

Definition 6.1. (Context) A context is an n-tuple $c = \{(p_1, v_1), (p_2, v_2), ..., (p_n, v_n)\}$ where p_i is a context variable, v_i is its corresponding value and n is the number of context variables.

Definition 6.2. (Context-GUI event pair coverage) A context-GUI event pair (c_i, e_j) is covered in a test suite T if there is at least one test case in T that executes GUI event e_j in context c_i.

Definition 6.3. (Context-state pair coverage) A context-state pair (c_i, s_j) is covered in a test suite T if for every GUI event e available in GUI state s_j, there is a test case in T that executes e in context c_i.

6.3. Test Suite Construction Framework

The framework uses an *event extraction cycle* to iteratively select and execute events from the GUI of the application under test to construct test cases one-event-at-a-time.

Algorithm 8 shows pseudocode for the test suite construction framework. It provides a high-level description of a process to automatically construct test suites for context sensitive Android apps. Lines 9-15 represent the event extraction cycle that incrementally constructs each test case. The framework requires specifications for several parameters (shown in boxes) to instantiate different test generation techniques. The test generation process consists of the following steps:

Step 1: Generate context covering array. Line 1 generates a covering array C from the combinatorial context model M specified as input. The covering array specifies a set of contexts that will be used to test the AUT. Each context specified in the covering array has a corresponding context event that changes the operating context of the AUT. This step occurs once for a single test suite.

Step 2: Initialize test case. Lines 4-8 initialize each test case in the test suite. Line 4 creates an empty event sequence. The *InitialContextStrategy* procedure call on line 5 uses a predefined strategy to select a start context for each test case and line 6 adds the associated

Algorithm 8: Pseudocode for test suite construction framework (boxes indicate framework parameters)

```
   Input   : android application package, AUT
   Input   : combinatorial context model, M
   Output: test suite, T
1  C ← generate context covering array from M
2  T ← φ
3  repeat
4  |  t_i ← φ
5  |  c_curr ← [InitialContextStrategy(C)]
6  |  add initial context event to test case t_i
7  |  install and launch AUT, add launch event to t_i
8  |  s_curr ← initial GUI state
9  |  while [TerminationCriterion] is not satisfied do
10 |  |  E_all ← GUI events in current GUI state s_curr
11 |  |  e_sel ← [EventSelectionStrategy(E_all, C_all, C)]
12 |  |  t_i ← t_i ∪ {e_sel}
13 |  |  s_curr ← current GUI state
14 |  |  c_curr ← current context
15 |  end
16 |  T ← T ∪ {t_i}
17 |  finalize test case (clear cache/SD card, uninstall app, etc.)
18 until [CompletionCriterion] is satisfied
```

context event as the first event in the new test case. Line 7 launches the AUT in the selected start context and adds a launch event to the test case. Line 8 retrieves the initial GUI state of the AUT.

Step 3: Select and execute an event. The *EventSelectionStrategy* procedure call on line 11 uses a predefined strategy to select and execute a context event or GUI event in each iteration of the event extraction cycle (lines 9-15). Event execution often changes the GUI state of the AUT and/or the value of one or more context variables. This iterative event selection and execution incrementally constructs a test case that may include context events and GUI events. A single test case ends when the algorithm satisfies a predefined

TerminationCriterion. We describe our pairwise event selection strategy in Section 6.3.1.

Step 4: Finalize test case. At the end of each test case, line 17 resets the state of the AUT and clears all data that may affect the outcome of subsequent test cases.

The algorithm generates multiple test cases until it satisfies a predefined *Completion-Criterion* that specifies when the test suite is complete.

6.3.1. Pairwise Event Selection

In each iteration of the event extraction cycle, the *EventSelectionStrategy* parameter in our framework specifies a strategy for choosing: (i) whether to execute a GUI event or context event and (ii) which particular event to execute, given a set of available GUI events and a context covering array.

Algorithm 9: Pairwise event selection

```
Input  : set of GUI events in current GUI state, E_all
Input  : context covering array, C
Output: GUI event, e_sel, or context event c_sel
 1 c_curr ← current context
 2 s_curr ← current GUI state
 3 e_sel ← select GUI event e_i from E_all such that (c_curr, e_i) is not yet covered
 4 if e_sel = φ then
 5 |   mark context-state pair (c_curr, s_curr) as covered
 6 |   c_sel ← select a context event for c_i ∈ C such that (c_i, s_curr) is not yet covered
 7 |   if c_sel ≠ φ then
 8 |   |   execute context event c_sel
 9 |   |   return c_sel
10 |   else
11 |   |   e_sel ← select random GUI event e_i from E_all
12 |   end
13 end
14 execute event e_sel
15 mark context-GUI event pair (c_curr, e_sel) as covered
16 return e_sel
```

Algorithm 9 shows pseudocode for our pairwise event selection strategy. The algorithm maintains a set of covered context-GUI event pairs (definition 6.2) and a set of covered context-state pairs (definition 6.3). The set of covered context-GUI event pairs enables the

algorithm to track which GUI events have been executed in a particular context. The set of covered context-state pairs enables the algorithm to recognize when all GUI events in a particular GUI state have been executed in a particular context. The pairwise event selection strategy prioritizes execution of GUI events in new contexts and enables generation of test cases that may contain multiple context changes interleaved with GUI events. Lines 1 and 2 identify the current context c_{curr} and current GUI state s_{curr} respectively. On line 3, the algorithm attempts to select and execute a GUI event e_i that has not yet been executed in the current context. If the algorithm finds a GUI event e_i that satisfies the criterion, it simply executes e_i. Failure to find such an event in the current GUI state s_{curr} indicates that all events available in s_{curr} have been executed in the current context c_{curr}. In this situation, the algorithm marks the context-state pair (c_{curr}, s_{curr}) as covered and attempts to find a context c_i in the covering array C such that there is at least one event in the current GUI state s_{curr} that has not been executed in c_i. If the algorithm finds such a context c_i, it executes a context event that changes the current context to c_i. If the algorithm is unable to find a GUI event or context event that satisfies any of the aforementioned criteria, it selects a GUI event uniformly at random.

6.3.2. Test Generation Techniques

	FixedContext	**RandStart**	**IterativeStart**	**RandInterleaved**	**PairsInterleaved**
InitialContextStrategy	Fixed	Random	Iterative	Random	Iterative
EventSelectionStrategy	Random	Random	Random	RandomInterleaved	PairwiseInterleaved
TerminationCriterion	Probabilistic	Probabilistic	Probabilistic	Probabilistic	Probabilistic
CompletionCriterion	Time	Time	Time	Time	Time

Table 6.3. Test generation techniques with corresponding parameter specifications

Table 6.3 shows five test generation techniques and their corresponding parameter-values in our framework. The *FixedContext* technique generates test suites in a single predefined context without any consideration for context changes. The *RandStart* and *IterativeStart* techniques generate test suites that execute context changes only at the beginning of each test case. The *RandStart* technique randomly selects a start context for each test case while the *IterativeStart* technique selects a start context for each test case in a round-robin

manner. With *RandStart* and *IterativeStart*, each test case begins with a single context event and continues with a sequence of randomly selected GUI events. The *RandInterleaved* technique generates test suites with randomly interleaved context events and GUI events. The *PairsInterleaved* technique uses our pairwise event selection technique to systematically interleave context events with GUI events. Only the *RandInterleaved* and *PairsInterleaved* techniques generate test cases that may contain several context changes. All the techniques defined in Table 6.3 except *FixedContext* enable execution of GUI events in multiple contexts within a single test suite. Each technique uses a predefined probability value (e.g. 0.05) to pseudorandomly terminate test cases and a predefined time limit to determine when test suite construction is complete.

6.3.3. Framework Implementation

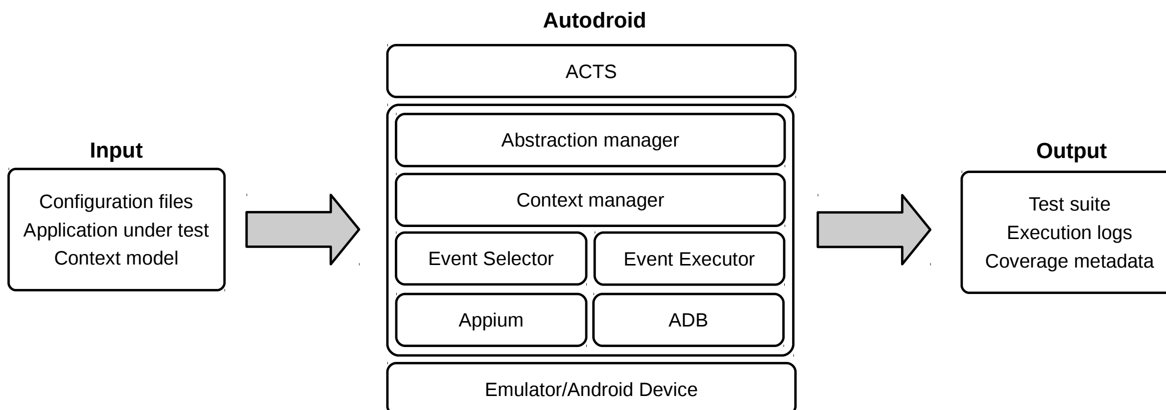


FIGURE 6.1. Framework implementation

Figure 6.1 shows the components of our framework. We implement the framework as part of our automated testing tool, Autodroid. The framework takes an Android application package (APK), combinatorial context model and configuration file as input. The combinatorial context model specifies a set of context variables and values for test generation. The configuration file enables users to specify different criteria for test suite construction. The framework uses the criteria specified in the configuration file to instantiate various test generation techniques. Autodroid automatically generates event sequence test suites, execution logs and code coverage metadata without need for source code or static abstract

models of the AUT. Autodroid uses ACTS[1] to generate covering arrays from combinatorial context models. It uses Appium[2] to identify and execute GUI events on an Android emulator or physical device. The Android Debug Bridge (ADB)[3] enables Autodroid to modify context variables and execute context changes. The *ContextManager* tracks the current operating context of the AUT during test generation. The *AbstractionManager* creates and manipulates event, action, GUI state and context abstractions. The *EventSelector* uses the abstractions to choose events for the *EventExecutor* to execute.

6.4. Experiments

6.4.1. Research Questions

Our experiments address the following research questions:

- **RQ1:** Does execution of GUI events in multiple contexts improve code coverage of test suites?
- **RQ2:** Does execution of GUI events in multiple contexts improve fault detection effectiveness of test suites?

6.4.2. Subject Application

We use an Android app called *EmployeeBase* to demonstrate the feasibility of our framework and combinatorial-based technique. We use the app as a case study of context-sensitive behavior that may require specialized testing techniques. The app has four activities and 1081 source lines of code (SLOC). It provides a GUI to create, retrieve and store employee information in a remote database. The app relies on an internet connection for most of its functionality. It polls the remote database for updates at a predefined interval that is determined by the current battery level and whether the device is connected to a power source. The app automatically adjusts the poll interval whenever it senses a change in the battery status (low/ok) or power status (connected/disconnected). It also notifies users whenever it is unable to retrieve records from the remote database.

[1] http://csrc.nist.gov/groups/SNS/acts/index.html

[2] http://appium.io

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

6.4.3. Experimental Setup

We generated 10 test suites with each technique and specified a fixed time limit of two hours (120 minutes) for each test suite. The techniques used are described as follows:

- The **FixedContext** technique generates a test suite in a single predefined context by randomly selecting and executing GUI events without consideration for context changes.
- The **RandStart** technique randomly selects a context event only at the beginning of each test case.
- The **IterativeStart** technique selects a different context event at the beginning of each test case by iterating through the context covering array in a round-robin manner.
- The **RandInterleaved** technique generates test suites with a random mix of context events and GUI events.
- The **PairsInterleaved** technique uses our pairwise event selection strategy to systematically execute GUI events in multiple contexts.

We used the *FixedContext* technique to construct test suites in a single context $c = \{$*WiFi=connected, Battery=OK, AC Power=connected, ScreenOrientation=Portrait*$\}$ that represents favorable operating conditions for the AUT. For the *RandStart*, *IterativeStart*, *RandInterleaved* and *PairsInterleaved* strategies, we generated a 2-way context covering array from a combinatorial context model. The combinatorial context model has four context variables $\{$*WiFi, Battery, AC Power, ScreenOrientation*$\}$ and two values for each variable (as described in section 6.1). We generated the test suites on Android 4.4 emulator instances and used a probability value of 0.05 to pseudorandomly terminate test cases. The Android emulator instances in our experiments only have Internet access via an emulated WiFi connection. Thus, any contexts with $\{$*WiFi=disconnected*$\}$ imply a loss of Internet access.

	FixedContext	RandStart	IterativeStart	RandInterleaved	PairsInterleaved
Average	77.04	83.56	83.56	78.81	**87.31**
Median	77.04	83.56	83.56	78.71	**87.21**
Minimum	77.04	83.52	83.52	70.18	**85.35**
Maximum	77.04	83.56	83.56	86.79	**90.03**
Standard Dev.	0	0.01	0.01	5.09	1.28

TABLE 6.4. Summary block coverage statistics across 10 test suites for each technique

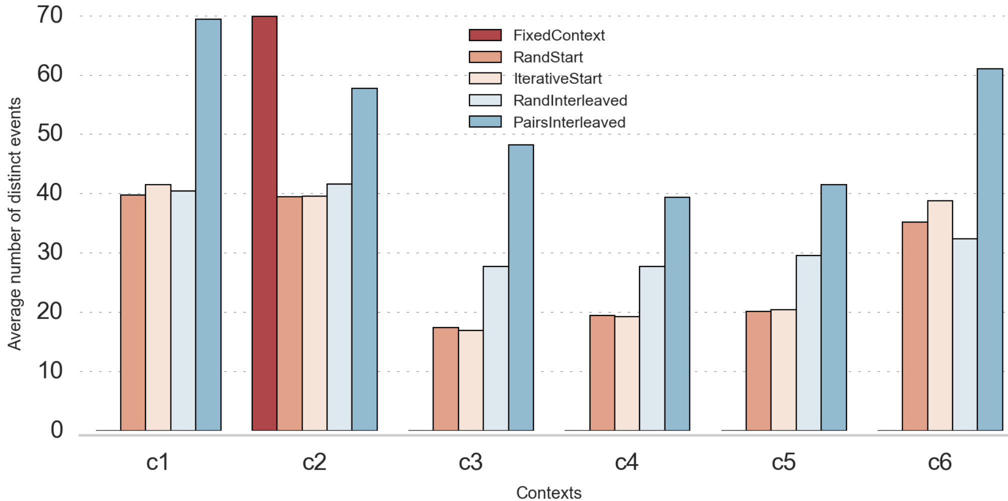


FIGURE 6.2. Average number of events executed in each context

6.4.4. Results and Discussion

Code coverage. Table 6.4 shows summary block coverage statistics for the *FixedContext*, *RandStart*, *IterativeStart*, *RandInterleaved* and *PairsInterleaved* techniques. The *FixedContext* technique executes randomly selected GUI events in a single predefined context. All 10 test suites generated with the *FixedContext* technique never achieve beyond 77.04% block coverage. Figure 6.2 shows the average number of distinct events executed in each of the six contexts defined in our context covering array. The *FixedContext* technique executes the highest number of distinct events in context c_2 and does not execute events in any other context. The *FixedContext* technique limits test suites to the subset of the AUT's code that is reachable in the predefined context and thus achieves the lowest average code coverage compared to the other techniques. Test suites that execute GUI events in only one context may be unable to test context-sensitive behavior that occurs when GUI events are executed

in multiple contexts.

The *RandStart* and *IterativeStart* techniques change the operating context of the AUT only at the beginning of each test case. This enables a single test suite to execute GUI events in multiple contexts. The *RandStart* and *IterativeStart* techniques achieve up to 7% higher code coverage on average compared to the *FixedContext* technique. These techniques achieve higher code coverage than the *FixedContext* strategy because they enable testing of context-sensitive behavior that occurs in response to GUI events executed in different operating contexts. These results suggest that execution of GUI events in multiple contexts may lead to increased code coverage for context-sensitive Android apps compared to techniques that execute GUI events in a single context.

The *RandInterleaved* technique generates test suites that contain a random mix of context events and GUI events in each test case. The test suites execute GUI events in multiple contexts and may contain multiple context changes in a single test case. The *RandInterleaved* technique shows a wide variation in code coverage across multiple test suites because it tends to execute too many context events at the expense of GUI exploration and code coverage. This may explain why the *RandStart* and *IterativeStart* techniques achieve higher code coverage on average compared to the *RandInterleaved* technique.

The *PairsInterleaved* technique, which uses our pairwise event selection strategy, achieves the highest code coverage compared to the other techniques in our experiments. Like the *RandInterleaved* technique, the *PairsInterleaved* technique may change the AUT's operating context multiple times within a single test case, but it regulates the frequency of context changes in relation to GUI events to avoid adverse effects on GUI exploration and code coverage. Inclusion of multiple context events in a single test case enables testing of context-sensitive behavior that is triggered by a specific sequence of context changes interleaved with GUI events. Figure 6.2 shows that the *PairsInterleaved* technique executes a higher number of GUI events in different contexts compared to the other techniques in our experiments. These results show that the code coverage of test suites may improve with test generation techniques that systematically execute GUI events in multiple contexts, allow

multiple context changes within a single test case and regulate the frequency of context changes in relation to GUI events.

	FixedContext	RandStart	IterativeStart	RandInterleaved	PairsInterleaved
java.lang.IllegalArgumentException	✗	✗	✗	✓	✓
libcore.io.GaiException	✗	✗	✗	✓	✓
java.lang.UnknownHostException	✗	✗	✗	✓	✓

TABLE 6.5. Exceptions found by each test generation technique

Fault detection. We examined execution logs from our experiments for exceptions/stack traces thrown by the AUT. We consider these exceptions to be indicators of faulty behavior in the AUT. Table 6.5 shows the exceptions found by the test suites generated with each technique. The *RandInterleaved* and *PairsInterleaved* techniques found three unique exception types across 10 test suites. The *FixedContext*, *RandStart* and *IterativeStart* techniques produce test suites that do not trigger any exceptions. The two techniques that enable multiple context changes within a single test case, *RandInterleaved* and *PairsInterleaved*, produce test suites that detect three types of exceptions in the application under test: *java.lang.IllegalArgumentException*, *libcore.io.GaiException* and *java.lang.UnknownHost- Exception*. These exceptions are related to areas of the AUT's code that depend on an Internet connection to retrieve data from a remote database. We observe that *java.lang.IllegalArgument-Exception* causes the AUT to crash when a test case changes the screen orientation, disconnects from WiFi and repeatedly presses the "back" button. Recall that the Android emulators in our experiments rely on a WiFi connection for Internet access. The other exceptions, *libcore.io.GaiException* and *java.lang.UnknownHostException*, occur when a test case executes a GUI event to retrieve data from the remote database but disconnects from WiFi before the connection is complete. These observations suggest that several faults in context-sensitive Android applications may be triggered by interactions between several context variables and GUI events. The fault detection effectiveness of test suites improves with test generation techniques that execute GUI events in multiple contexts and allow multiple context changes within a single test case.

6.4.5. Threats to Validity

The primary threat to validity of our experiments is the generalizability of our results. The characteristics of the application under test may impact the effectiveness of our framework and event selection technique. Another possible threat is the number of context variables incorporated into the test cases. When and how often to change the context variables is still unknown for optimal cost effectiveness. In future work, we will extend our empirical study to include several context variables and Android apps of varying size and complexity. The randomized nature of the techniques in the study is another threat to validity. To minimize this threat, we constructed 10 test suites with each test generation technique.

6.5. Summary and Conclusions

Mobile applications may react to context events in addition to GUI events. Context events may alter the operating context of an application under test and cause changes in behavior. It is important to generate tests that manipulate the operating context of the AUT to test context-sensitive behavior. This chapter describes a context-aware automated testing framework that allows testers to use different criteria to construct Android application test suites with context events and GUI events. As part of our framework, we develop a pairwise event selection technique that systematically executes GUI events in multiple contexts to test context-sensitive behavior. The results of our experiments show that our pairwise technique improves fault detection effectiveness and achieves up to 10% higher code coverage compared to a technique that generates test suites in a single predefined context. This chapter demonstrates the importance of manipulating the operating context of an AUT during test generation and shows that systematic execution of GUI events in multiple contexts may improve the code coverage and fault detection effectiveness of test suites for context-sensitive Android applications.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

Prior work in automated GUI testing focuses on model-based techniques that require a preexisting abstract model of the AUT for offline generation and subsequent execution of event sequences. Several studies investigate online GUI testing as an alternative to model-based testing and often use algorithms that select and execute events uniformly at random. The majority of online GUI testing tools, techniques and experiments in prior work are often not directly applicable to mobile applications without significant modification. The majority of prior work in mobile application testing focuses predominantly on GUI events and does not describe techniques to test context-sensitive behavior triggered by context changes (e.g. changes in network connectivity, battery levels, screen orientation, etc.). In many cases, existing tools do not produce event sequences that can be reused for automated regression testing and reproduction of failures.

This dissertation presents novel probabilistic and combinatorial-based algorithms for online construction of Android application test suites. Our algorithms are based on an event extraction cycle that iteratively identifies, selects and executes events to construct reusable test cases one-event-at-a-time without need for source code analysis or preexisting abstract models of the AUT. We implement our algorithms in an automated GUI testing tool called Autodroid and perform empirical studies to assess the effectiveness of our techniques. The results of our experiments provide empirical data that may help software testing researchers and practitioners choose between several online test generation techniques for mobile applications.

7.1. Summary of Contributions

The following are the major contributions of this dissertation:

Autodroid. We develop an automated GUI testing tool, Autodroid, that implements an online approach to automatic construction of Android application test suites. Autodroid's online algorithms consist of an event extraction cycle that iteratively identifies, selects and

executes events to construct event sequences one-event-at-a-time. Autodroid generates test cases that conform to an event sequence metamodel to enable reuse for automated regression testing and reproduction of failures. We demonstrate that Autodroid's random-based algorithm generates test suites that achieve higher code coverage than test suites generated with Monkey, a widely used random GUI testing tool for Android applications. We implemented the algorithms and techniques in this work as part of Autodroid.

Frequency-based test suite construction. We developed an online test suite construction technique that uses the prior execution frequency of events to prioritize selection of previously unexecuted events and minimize redundant event execution during test suite construction. We develop two frequency-based event selection algorithms that alter event selection probabilities based on the prior execution frequency of available events in a given GUI state. We compared test suites generated with our frequency-based event selection algorithms to test suites generated with uniform random event selection. The major findings from our experiments with seven Android applications are as follows:

- The event selection strategy used in an online test suite construction algorithm has a significant impact on cost-effectiveness in terms of code coverage and event coverage.
- Given a fixed time budget of two hours, our frequency-based event selection algorithms generate test suites that achieve higher code coverage and event coverage compared to test suites generated with uniform random event selection.
- Test suite construction algorithms that prioritize selection of previously unexecuted events tend to achieve higher code coverage and event coverage compared to random-based algorithms that select and execute events uniformly at random.

Combinatorial-based test suite construction. The order in which events occur may influence the behavior of Event Driven Software (EDS) including GUI-based software such as mobile applications. We developed a combinatorial-based test suite construction technique that considers potential interactions between events that occur in a particular order. During test suite construction, our combinatorial-based algorithm prioritizes selection of

previously unexecuted events, considers the order in which events have previously occurred and maximizes coverage of n-way event combinations, where n is a specified event combination strength. We compared 2-way and 3-way combinatorial test suites to random-based test suites in terms of code coverage and event coverage. The major findings from our experiments with seven Android applications are as follows:

- Given a fixed time budget of two hours, our 2-way and 3-way combinatorial-based test suites achieve higher code coverage and event coverage than a random-based technique despite the additional overhead of computing event combinations.
- The cost-effectiveness of higher event combination strengths ($n > 2$) depends on the characteristics of the AUT. Compared to 2-way combinatorial-based test suites, higher event combination strengths ($n > 2$) are likely to be most effective for simple Android applications with a small number of activities.
- Online test suite construction with event combination strength $n > 2$ may require additional time to generate test suites that are more cost-effective than those generated with lower event combination strengths because of the computational cost of computing event combinations.

Framework for testing context-sensitive Android applications. Mobile applications may react to context events (e.g. changes in network connectivity) in addition to GUI events. Context events may alter the operating context of an application under test and cause changes in behavior. We develop a framework that allows testers to use different criteria to automatically construct test suites for context-sensitive Android applications and a pairwise event selection technique to systematically execute GUI events in multiple contexts. The framework combines context manipulation and test generation into a single process. We use the framework to instantiate multiple test generation techniques that integrate context events and GUI events in different ways and perform an empirical study to compare test suites generated with the different techniques. The major findings from our experiments with a context-sensitive Android application are as follows:

- Test generation techniques that execute GUI events in multiple contexts are nec-

essary to improve the effectiveness of test suites for context-sensitive mobile applications. All such techniques must regulate the frequency of context changes in relation to GUI events and minimize the combinatorial explosion that occurs when combining context events and GUI events.

- Our pairwise event selection technique achieves significant improvement in code coverage and fault-finding effectiveness compared to a technique that selects a random mix of GUI events and context events, techniques that change the operating context of the AUT only at the start of each test case and a technique that randomly executes GUI events in a single predefined context.

7.2. Future Work

This dissertation lays the foundation for future work in the following areas.

Context-sensitivity in mobile applications. Mobile applications may react to a number of context events in addition to GUI events generated by users. This represents an increase in the input space that must be sampled in order to effectively test mobile applications, especially those that rely on contextual information to provide context-sensitive behavior to users. This dissertation proposes a framework that automatically constructs test suites for context-sensitive Android application. We will extend our empirical study to include additional subject applications and improve our techniques to consider constraints between context variables based on real-world data collected from users. We will also investigate the impact of higher interaction strengths between context variables on the code coverage and fault-finding effectiveness of test suites generated with our framework.

Reinforcement learning and online graph exploration techniques. Online GUI testing algorithms traverse the GUI of an application under test by visiting GUI states and selecting events to execute. This process is similar to online learning and online graph exploration problems. Online GUI testing algorithms may benefit from reinforcement learning techniques that enable more intelligent event selection based on information about the AUT's response to previously executed events. One major challenge in this research direction is the definition of a suitable reward function that effectively embodies the goals of the testing pro-

cess. This is particularly challenging since the behavior of software differs from traditional reinforcement learning environments in many ways.

Fault detection studies. This dissertation evaluates online test suite construction techniques in terms of code coverage and event coverage. We will perform additional empirical studies that focus on the fault finding effectiveness of our techniques. This may require manual fault seeding of subject applications or the use of mutation testing tools.

Extensive tool support. Our automated GUI testing tool, Autodroid, provides tool support for the techniques and empirical studies in this work. We will extend Autodroid with test debugging, test prioritization and test reduction capabilities to support further research in those areas.

REFERENCES

[1] 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*. ACM, 2015, pp. 83–93.

[2] D. Amalfitano, N. Amatucci, A. R. Fasolino, P. Tramontana, E. Kowalczyk, and A. M. Memon, "Exploiting the saturation effect in automatic random testing of android applications," in *2015 2nd ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 2015, pp. 33–43.

[3] D. Amalfitano, N. Amatucci, A. M. Memon, P. Tramontana, and A. R. Fasolino, "A general framework for comparing automatic testing techniques of android mobile apps," *Journal of Systems and Software*, vol. 125, pp. 322–343, 2017.

[4] D. Amalfitano, A. R. Fasolino, and P. Tramontana, "A GUI crawling-based technique for android mobile application testing," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2011, pp. 252–261.

[5] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "MobiGUITAR: Automated model-based testing of mobile apps," *IEEE Software*, vol. 32, no. 5, pp. 53–59, 2015.

[6] A. Arcuri and L. Briand, "Adaptive random testing: An illusion of effectiveness?" in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 2011, pp. 265–275.

[7] ——, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 2011, pp. 1–10.

[8] ——, "Formal analysis of the probability of interaction fault detection using random testing," *IEEE Transactions on Software Engineering*, vol. 38, no. 5, pp. 1088–1099, 2012.

[9] A. Arcuri, M. Z. Iqbal, and L. Briand, "Random testing: Theoretical results and practical implications," *IEEE Transactions on Software Engineering*, vol. 38, no. 2, pp. 258–277, 2012.

[10] G. Bae, G. Rothermel, and D.-H. Bae, "Comparing model-based and dynamic event-extraction based GUI testing techniques: An empirical study," *Journal of Systems and Software*, vol. 97, pp. 15–46, 2014.

[11] F. Belli, "Finite state testing and analysis of graphical user interfaces," in *Proceedings of the 12th International Symposium on Software Reliability Engineering.* IEEE, 2001, pp. 34–43.

[12] M. Böhme and S. Paul, "On the efficiency of automated testing," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering.* ACM, 2014, pp. 632–642.

[13] R. C. Bryce and C. J. Colbourn, "Prioritized interaction testing for pair-wise coverage with seeding and constraints," *Information and Software Technology*, vol. 48, no. 10, pp. 960–970, 2006.

[14] ——, "The density algorithm for pairwise interaction testing," *Software Testing Verification and Reliability*, vol. 17, no. 3, pp. 159–182, 2007.

[15] R. C. Bryce, C. J. Colbourn, and M. B. Cohen, "A framework of greedy methods for constructing interaction test suites," in *Proceedings of the 27th international conference on Software engineering.* ACM, 2005, pp. 146–155.

[16] R. C. Bryce and A. M. Memon, "Test suite prioritization by interaction coverage," in *Workshop on Domain-Specific Approaches to Software Test Automation.* ACM, 2007, pp. 1–7.

[17] S. Carino, "Dynamically testing graphical user interfaces," Ph.D. dissertation, The University of Western Ontario, 2016.

[18] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. Tse, "Adaptive random testing: The art of test case diversity," *Journal of Systems and Software*, vol. 83, no. 1, pp. 60–66, 2010.

[19] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android:

Are we there yet?" in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 2015, pp. 429–440.

[20] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The aetg system: An approach to testing based on combinatorial design," *IEEE Transactions on Software Engineering*, vol. 23, no. 7, pp. 437–444, 1997.

[21] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton, "The combinatorial design approach to automatic test generation," *IEEE software*, vol. 13, no. 5, pp. 83–88, 1996.

[22] M. B. Cohen, M. B. Dwyer, and J. Shi, "Interaction testing of highly-configurable systems in the presence of constraints," in *Proceedings of the 2007 international symposium on Software testing and analysis.* ACM, 2007, pp. 129–139.

[23] G. A. Di Lucca and M. Di Penta, "Considering browser interaction in web application testing," in *2003 Fifth IEEE International Workshop on Web Site Evolution.* IEEE, 2003, pp. 74–81.

[24] J. W. Duran and S. C. Ntafos, "An evaluation of random testing," *IEEE Transactions on Software Engineering*, no. 4, pp. 438–444, 1984.

[25] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE Transactions on Software Engineering*, vol. 28(2), no. 2, pp. 159–182, 2002.

[26] emma.sourceforge.net, "What is block coverage?" http://emma.sourceforge.net/faq.html#q.blockcoverage, 2006, (Accessed: 16-05-2017).

[27] Entrepreneur.com, "By 2017, the app market will be a $77 billion industry (infographic)," https://www.entrepreneur.com/article/236832, 2014, (Accessed: 10-25-2016).

[28] Google, "Espresso - google," https://google.github.io/android-testing-support-library/docs/espresso/, (Accessed: 01-13-2017).

[29] ——, "UI/application exerciser monkey," https://developer.android.com/studio/test/monkey.html, (Accessed: 2017-01-28).

[30] M. Grindal, J. Offutt, and S. F. Andler, "Combination testing strategies: a survey," *Software Testing, Verification and Reliability*, vol. 15(3), no. 3, pp. 167–199, 2005.

[31] R. Hamlet, "Random testing," *Encyclopedia of software Engineering*, 1994.

[32] C. Hu and I. Neamtiu, "Automating GUI testing for android applications," in *Proceedings of the 6th International Workshop on Automation of Software Test*. ACM, 2011, pp. 77–83.

[33] IDC Research, "Smartphone OS market share, 2016 q2," http://www.idc.com/prodserv/smartphone-os-market-share.jsp, 2016, (Accessed: 10-25-2016).

[34] JUnit, "Junit," http://junit.org, (Accessed: 01-18-2017).

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

[36] D. R. Kuhn, J. M. Higdon, J. F. Lawrence, R. N. Kacker, and Y. Lei, "Combinatorial methods for event sequence testing," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 2012, pp. 601–609.

[37] D. R. Kuhn and M. J. Reilly, "An investigation of the applicability of design of experiments to software testing," in *Proceedings of the 27th Annual NASA Goddard/IEEE Software Engineering Workshop*. IEEE, 2002, pp. 91–95.

[38] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, "Software fault interactions and implications for software testing," *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 418–421, 2004.

[39] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing," *Software Testing, Verification and Reliability*, vol. 18, no. 3, pp. 125–148, 2008.

[40] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Transactions on Software Engineering*, vol. 33(4), no. 4, pp. 225–237, 2007.

[41] Z. Liu, X. Gao, and X. Long, "Adaptive random testing of mobile application," in *2010 2nd International Conference on Computer Engineering and Technology (ICCET)*, vol. 2. IEEE, 2010, pp. V2–297.

[42] 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*. ACM, 2013, pp. 224–234.

[43] 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*. ACM, 2014, pp. 599–609.

[44] T. A. Majchrzak and M. Schulte, "Context-dependent testing of applications for mobile devices," *Open Journal of Web Technologies (OJWT)*, vol. 2, no. 1, pp. 27–39, 2015.

[45] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The Annals of Mathematical Statistics*, vol. 18, no. 1, pp. 50–60, 1947.

[46] 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*. ACM, 2016, pp. 94–105.

[47] A. Memon, I. Banerjee, B. N. Nguyen, and B. Robbins, "The first decade of GUI ripping: Extensions, applications, and broader impacts," in *2013 20th Working Conference on Reverse Engineering*. IEEE, 2013, pp. 11–20.

[48] A. M. Memon, "An event-flow model of gui-based applications for testing," *Software Testing Verification and Reliability*, vol. 17, no. 3, pp. 137–158, 2007.

[49] A. M. Memon, I. Banerjee, and A. Nagarajan, "Gui ripping: Reverse engineering of graphical user interfaces for testing." in *2003 Working Conference on Reverse Engineering*, vol. 3, 2003, p. 260.

[50] A. M. Memon and B. N. Nguyen, "Advances in automated model-based system testing of software applications with a GUI front-end," *Advances in Computers*, vol. 80, pp. 121–162, 2010.

[51] A. Mesbah, A. Van Deursen, and S. Lenselink, "Crawling ajax-based web applications through dynamic analysis of user interface state changes," *ACM Transactions on the Web (TWEB)*, vol. 6, no. 1, p. 3, 2012.

[52] B. P. Miller, G. Cooksey, and F. Moore, "An empirical study of the robustness of MacOS applications using random testing," in *Proceedings of the 1st international workshop on Random testing*. ACM, 2006, pp. 46–54.

[53] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek, "Reducing combinatorics in GUI testing of android applications," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 559–570.

[54] I. B. Mohamad and D. Usman, "Standardization and its effects on k-means clustering algorithm," *Research Journal of Applied Sciences, Engineering and Technology*, vol. 6, no. 17, pp. 3299–3303, 2013.

[55] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk, "Automatically discovering, reporting and reproducing android application crashes," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2016, pp. 33–44.

[56] I. C. Morgado and A. C. Paiva, "The iMPAcT tool: Testing UI patterns on mobile applications," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 876–881.

[57] B. N. Nguyen, B. Robbins, I. Banerjee, and A. Memon, "GUITAR: an innovative tool for automated testing of gui-driven software," *Automated Software Engineering*, vol. 21, no. 1, pp. 65–105, 2014.

[58] C. D. Nguyen, A. Marchetto, and P. Tonella, "Combining model-based and combinatorial testing for effective test case generation," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, 2012, pp. 100–110.

[59] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Computing Surveys (CSUR)*, vol. 43, no. 2, p. 11, 2011.

[60] Oracle Corporation, "javax.swing (java platform se 7)," https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html, (Accessed: 01-21-2017).

[61] R. Recorder, "Robotium reccorder - robotium tech," https://robotium.com/products/robotium-recorder, (Accessed: 06-17-2017).

[62] Robotium, "Robotiumtech/robotium: Android ui testing," https://github.com/RobotiumTech/robotium, (Accessed: 01-13-2017).

[63] R. Sharma, M. Gligoric, A. Arcuri, G. Fraser, and D. Marinov, "Testing container classes: Random or systematic?" *Fundamental Approaches to Software Engineering*, pp. 262–277, 2011.

[64] K. Song, A. R. Han, S. Jeong, and S. Cha, "Generating various contexts from permissions for testing android applications," in *27th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, 2015, pp. 87–92.

[65] T. Su, "Fsmdroid: guided gui testing of android apps," in *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, 2016, pp. 689–691.

[66] TechCrunch.com, "Users have low tolerance for buggy apps only 16% will try a failing app more than twice," https://techcrunch.com/2013/03/12/users-have-low-tolerance-for-buggy-apps-only-16-will-try-a-failing-app-more-than-twice/, 2013, (Accessed: 10-25-2016).

[67] P. Thévenod-Fosse and H. Waeselynck, "An investigation of statistical software testing," *Software Testing, Verification and Reliability*, vol. 1, no. 2, pp. 5–25, 1991.

[68] W. Wang, Y. Lei, S. Sampath, R. Kacker, R. Kuhn, and J. Lawrence, "A combinatorial approach to building navigation graphs for dynamic web applications," in *2009 IEEE International Conference on Software Maintenance*. IEEE, 2009, pp. 211–220.

[69] W. Wang, S. Sampath, Y. Lei, and R. Kacker, "An interaction-based test sequence generation approach for testing web applications," in *2008 11th IEEE High Assurance Systems Engineering Symposium*. IEEE, 2008, pp. 209–218.

[70] L. White, H. Almezen, and N. Alzeidi, "User-based testing of GUI sequences and their

interactions," in *Proceedings of the 12th International Symposium on Software Reliability Engineering.* IEEE, 2001, pp. 54–63.

[71] Q. Xie and A. M. Memon, "Studying the characteristics of a "good" gui test suite," in *17th International Symposium on Software Reliability Engineering.* IEEE, 2006, pp. 159–168.

[72] ——, "Using a pilot study to derive a GUI model for automated testing," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 18, no. 2, pp. 7:1–7:35, 2008.

[73] W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated GUI-model generation of mobile applications," in *International Conference on Fundamental Approaches to Software Engineering.* Springer, 2013, pp. 250–265.

[74] X. Yuan, M. Cohen, and A. M. Memon, "Covering array sampling of input event sequences for automated gui testing," in *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering.* ACM, 2007, pp. 405–408.

[75] X. Yuan, M. B. Cohen, and A. M. Memon, "GUI interaction testing: Incorporating event context," *IEEE Transactions on Software Engineering*, vol. 37, no. 4, pp. 559–574, 2011.

[76] X. Yuan and A. M. Memon, "Generating event sequence-based test cases using GUI runtime state feedback," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 81–95, 2010.

[77] R. N. Zaeem, M. R. Prasad, and S. Khurshid, "Automated generation of oracles for testing user-interaction features of mobile apps," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation.* IEEE, 2014, pp. 183–192.

[78] Y. Zhauniarovich, A. Philippov, O. Gadyatskaya, B. Crispo, and F. Massacci, "Towards black box testing of android apps," in *2015 10th International Conference on Availability, Reliability and Security (ARES).* IEEE, 2015, pp. 501–510.