AUTOMATED INPUT GENERATION TECHNIQUES FOR TESTING ANDROID APPLICATIONS

by

Nariman Mirzaei

A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
in Partial Fulfillment of
the Requirements for the Degree
of
Doctor of Philosophy
Computer Science

Committee:

____________________	Dr. Paul Ammann, Dissertation Director
____________________	Dr. Sam Malek, External Committee Member
____________________	Dr. Elizabeth White, Committee Member
____________________	Dr. Houman Homayoun, Committee Member
____________________	Dr. Thomas LaToza, Committee Member
____________________	Dr. Sanjeev Setia, Department Chair
____________________	Dr. Kenneth S. Ball, Dean, Volgenau School of Engineering
Date: ____________________	Summer Semester 2016 George Mason University Fairfax, VA

Automated Input Generation Techniques for Testing Android Applications

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

By

Nariman Mirzaei
Master of Science
Indiana University - Bloomington, 2009
Bachelor of Science
Amirkabir University of Technology, 2007

Director: Paul Ammann, Associate Professor
Department of Computer Science

Summer Semester 2016
George Mason University
Fairfax, VA

Dedication

To my father whose every sacrifice made this possible, and to my mother whose love and support are never ending.

Acknowledgments

I have been very fortunate to be in the company of great friends, colleagues, teachers and mentors throughout different stages of my life. From these people, I have learned lessons far more important than anything that can be thought by any book or in any classroom. Each person and every experience has helped shaping me into the person I am, and hopefully into a better one everyday. It is impossible to find the right words to express how I feel about these people and to show my gratitude toward every single one of them.

First, I must express my deepest gratitude to my advisor, Dr. Sam Malek for his guidance and support during the past few years. I cannot envision getting to the finish-line without your encouragement, constructive criticism, and tough love. Thank you for being patient, understanding and supportive of me during all my hardships. Thank you for bearing with me and not punching me in the face after my dumb mistakes. And finally thank you for being a great mentor, a good friend and a big brother for me.

I thank Dr. Paul Ammann for for administering the role of dissertation director after Sam joined University of California, Irvine. Not only his wisdom and support eased the way for my graduation, his calmness and cool manner always inspired me.

This dissertation was not possible without the guidance of both current and former members of my comprehensive exam and dissertation committees. I am grateful to all the members of my committees: Dr. Elizabeth White, Dr. Houman Homayou, and Dr. Thomas LaToza.

During my PhD journey, I enjoyed collaborating with a group of outstanding colleagues and friends. This includes Sam's research group members and alumni: Dr. Joshua Garcia, Dr. Hamid Bagheri, Dr. Naeem Esfahani, Dr. Ahmed Elkhodary, Dr. Riyadh Mahmood, Dr. Ehsan Kouroshfar, Alireza Sadeghi, and Reyhaneh Jabbarvand, and also members and alumni of software engineering lab: Dr. Nan Li, and Lin Deng. I learned a great deal from every single one of you. I truly enjoyed working alongside you and talking to you everyday. For that I am grateful to all of you.

Among the members of Sam's research group, Josh has been one person that I have worked with the most. Our ICSE 2016 paper was not even remotely possible without his help. Not only, he helped me with the writing in crunch-time, but also he helped with designing and running the experiments and developing new ideas. More importantly, he was always present as a friend that I could lean on during the down times of my PhD journey. He listened to me bragging for hours and hours, tried calming me down and always did his best to help with the situation. I feel privileged to have a friend like Josh that I could both work with and learn from.

During my years at George Mason, Ehsan has been someone that I have spent the most time with both on and off campus. Since the my first days at Mason, we have been sarcastically calling each other "Doctor", thinking that we would never graduate to become a doctor. After six years, it is both extremely flattering and surreal that we are both graduating around the same time. During this journey, he has been always there to calm me down during my frustrations, cheer me up in sadness and to accompany me in happy moments. I am immensely grateful for his friendship.

Throughout my academic life, I have been blessed to always have great teachers and professors. I never found the notion of class and coursework to be a useful learning tool and I always could learn the concepts on my own. Having said that, I strongly believe that there is not a single book that could teach me the lessons of life, inspire me to be a better person, and to show me how to develop my skills. These great men have had a great influence on me and I will always look up to them to be a better person: Mr. Faani my fifth grade teacher, Mr. Masoudnia my elementary school counselor , Late Mr. Noorbehbahani my high-school English teacher, Mr. Vaezi my high-school Vice-principal, and finally Dr. Ed Robertson my software engineering course professor at Indiana University.

I am forever grateful to my family. I am thankful to my parents, Mehrdad and Sussan, for all their sacrifices, love, and support. My father's main goal has always been to provide me and my brother with the opportunity to be successful. He has always been there for us as a person that we could lean on regardless of the situation, the true definition of a father figure, "the best friend". My mother, has been always supportive of me even-though she did not necessarily agree with all my decisions. For me she is the icon for true love, honesty and strength. I am thankful to my little brother, Navid, for taking care of my parents and being there for them while I was away from home for the past few years. I sincerely love you and I miss having you around all the time as my punching bag. I am immensely grateful to my late grandpa, Ahmad, and my grandma, Mahrokh, for their unconditional love and all the great memories throughout my life. I am also thankful to my uncle Sassan for his constant support since my first days in the U.S. Without the help of my family, I could not have started my PhD, let alone finish it.

I am grateful to my host-family in Bloomington, IN, Don and Judy Skirvin. I got to know Don and Judy through an amazing program that I signed up for during the orientation at Indiana University called *Bloomington World-wide Friendship.* I cannot write enough about their love, kindness and support during my years at IU. Not only, they filled the gap of not having any family around during my first days in U.S, they took me in as a family member into their lovely home. For that and all the great memories I am immensely grateful.

I am certainly the most fortunate to always have had great friends in my company. For that I am grateful to all my friends, and specially the members of gang of four plus one, Sadra, Amin, Mehdi and Rouzbeh. I am certainly not thankful of you guys teasing me about being a student for too long or constantly giving me a hard time with every other thing. But I want you all to know, that I realize how lucky I am to have such great friends whom I could always depend on for the past 20 years. There is nothing that I do not trust you guys with and I have truly enjoyed our time together.

Thank you all again for being part of my journey.

Table of Contents

List of Tables

List of Figures

Abstract

AUTOMATED INPUT GENERATION TECHNIQUES FOR TESTING ANDROID APPLICATIONS

Nariman Mirzaei, PhD

George Mason University, 2016

Dissertation Director: Dr. Paul Ammann

The rising popularity of Android and the GUI-driven nature of its apps have motivated the need for applicable automated testing techniques. This dissertation describes two automatic techniques for generating inputs for testing Android applications, SIG-Droid and TrimDroid. Both presented techniques employ a model-based approach to capture the event-driven nature of Android applications into two inferred models : *Interface Model* and *Activity Transition Model*. The Interface Model is used to find values that an app can receive through its interfaces. The Activity Transition Model is used to generate sequences of events that resemble user interactions with the app. SIG-Droid uses symbolic execution for obtaining test inputs that ensure covering each reachable branch in the program, while TrimDroid focuses on reducing the combinatorics (i.e. dealing with combinatorial explosion of test cases) in combinatorial testing of Android apps. TrimDroid relies on program analysis to extract formal specifications that express the dependencies between the GUI elements. The dependencies among the GUI elements comprising the app are used to reduce the number of combinations with the help of a solver. All conducted experiments corroborate the effectiveness and efficiency of SIG-Droid and TrimDroid.

Chapter 1: Introduction

Advances in mobile devices' processing power have resulted in an increase in both popularity of mobile devices and their capabilities. The current software distribution model for all existing mobile platforms is through the use of online app stores or markets. These online markets have made it very easy and cheap for the developers to produce apps that can reach a large number of consumers.

The mobile app markets are creating a paradigm shift in the way software is delivered to the end-users. The benefits of this software supply model are plenty, including the ability to rapidly and effectively introduce, maintain, and enhance software used by the consumers. By providing a medium for reaching a large consumer market at a nominal cost, app markets have leveled the software development field, allowing small entrepreneurs to compete head-to-head against prominent software development companies.

Android, introduced by Google in 2008, is currently one of the most popular available mobile platforms. It is a comprehensive software framework for mobile communication devices including smartphones and PDAs. Android has had a meteoric rise since its inception partly due to its vibrant app market that currently provisions nearly a million apps, with thousands added and updated on a daily basis [3].

Testing is traditionally a manual, expensive, and cumbersome process. While there are numerous existing methods for Unit Testing and GUI testing of Android apps there is a lack of a comprehensive technique for system testing of these apps. Not surprisingly there is an increasing demand by developers, consumers, and market operators for automated testing techniques applicable to Android apps. From a technical standpoint, a key obstacle is the lack of practical techniques to generate test inputs to test the apps submitted to the app markets.

In automated testing techniques, often test cases are generated using some information about the system under test that either already exists or is acquired from the program structure, source code, the specifications, execution logs and so on. Some of the well-known techniques that can be considered for generating test input include: (1) model-based test case generation; (2) symbolic execution; (3) combinatorial testing; (4) random and pseudo random testing [4].

Although, there is a wealth of research on using the aforementioned techniques in traditional software systems, but only a few works have attempted to investigate these techniques in mobile platforms such as in Android. Most of the recent testing approaches that target Android focus on either improving the state-of-the-art model-based techniques [5–7] or improving random testing coverage by employing a set of heuristics [8, 9].

While promising, model-based testing techniques have a major shortcoming. Model inference and model based testing help us understand and test the behaviors of an app by means of abstract models and the inferred models can be used to generate sequences of events that simulate the user's interaction with the app [10, 11]. To make these sequences executable, we need to accompany them with manually-defined or arbitrarily chosen values for data input widgets. But defining the input data manually is expensive, suffers from human mistakes and is hard to deal with in large models. Moreover, both approaches ignore the interaction between input widgets, reducing both the code coverage and the chances of detecting faults [12].

On the other hand, symbolic execution and combinatorial testing are two techniques that can be used for generating input values for data widgets [13–15], but neither can deal with the problem of exercising the sequences of events [16]. The underlying insight in this research is that a model-based approach can be combined with either symbolic execution or combinatorial testing to complement each other and compensate for each other's weaknesses. The inferred models can be used to analyze the behaviors of the app and to generate sequences of events for the test cases. Accordingly, symbolic execution and combinatorial testing are used to generate sets of input values to augment the sequences of events to generate concrete and executable test cases. Hence, the problem of testing Android applications is broken down into two problems of (1) generating sequences of events, where each sequence captures a particular use-case for the app, and (2) generating proper concrete values for GUI data widgets that take user inputs.

The main focus of this research is on the problem of system-level input generation for Android applications. To that end, this dissertation first presents a model-inference technique that leverages the knowledge of Android specifications to automatically extract two models from an app's APK file:[1] an *Activity Transition Model (ATM)*, representing the event-driven behavior of the app, and an *Interface Model (IM)*, representing all of the input interfaces in the app and the widgets they contain, including buttons, input boxes, etc. These models are used to guide the generation of event sequences aimed at simulating actual user behaviors.

This is followed by investigating symbolic execution as the main mechanism for generating concrete values to augment the sequences of events to generate system-level test cases. The proposed symbolic approach is built on top of NASA's Java Path Finder [17] to systematically generate inputs for Android apps that achieve high code coverage. It uses the *ATM* and the *IM* to exhaustively pinpoint possible ways an app can receive inputs. It then exchanges all concrete inputs with symbolic values, and gathers the constraints around those inputs.

Although the evaluation results support the effectiveness of the symbolic approach in terms of code coverage, extending it to support all real-world apps is not easy. Android apps are built using an application development framework (ADF), which allows the programmers to extend the base functionality of the platform using a well-defined API that includes more than 17000 Java classes [18]. Hence, efficient and automatic symbolic execution of Android apps requires an overwhelming engineering effort to support the API libraries of the Android ADF.

Combinatorial testing has shown to be a promising alternative for generating input

[1]An APK file is a Java byte-code package used to install Android apps.

values for data widgets in Android applications [19]. But exhaustive combinatorial GUI testing is often viewed to be impractical due to the explosion of possible combinations for even the smallest applications [14]. *T-way* combinatorial testing [15] is a heuristic-based alternative for exhaustive combinatorial testing that only considers a subset of GUI widgets (i.e., t) [20]. But arbitrary selection of t widgets to be combinatorily tested can result in less effective test-cases than an exhaustive approach in terms of both code coverage and fault detection [12].

To mitigate these issues, this dissertation proposes a novel combinatorial approach that employs static analysis techniques that are informed by the rules and constraints imposed by the Android ADF to identify GUI widgets that have dependencies on one another. Thus, the set of GUI widgets with dependencies become candidates for t-way combinatorial testing. Avoiding the generation of test cases for GUI widgets that do not have any dependencies significantly reduces the number of test cases. For identifying the dependencies, the interactions between the widgets and the variables in their *def-use* chain are statically analyzed. Finally, an efficient constraint solver is used to enumerate the test cases covering all possible interactions among the GUI widgets. Finally, the *ATM* and the *IM* are used to pinpoint possible ways an app can behave when it receives GUI inputs. These models are transformed into Alloy [21] specifications, the solutions to which are enumerated with a constraint solver for deriving the suite of test cases.

Despite the fact that combinatorial testing has shown to be effective in testing GUI application [15] by testing all combinations of values for input widgets of the program, breaking down the input domain of each input widget into classes is a challenge. Here, symbolic execution can be leveraged as a data generation technique to systematically derive the input classes for unbounded data-widgets in Android apps.

These approaches are extensively evaluated using several real-world open-source Android applications with respect to effectiveness in terms of code coverage, and efficiency in terms of reducing the number of generated test cases compared to an exhaustive approach. Moreover, the resulting code coverage is compared to several existing techniques.

The rest of this dissertation is organized as follows. Chapter 2 provides a background on Android, Symbolic Execution and Combinatorial Testing, followed by a discussion of the related work. Chapter 3 describes the problem and specifies the scope of this thesis. Chapter 4 describes the process for extraction of the required models. Chapter 5 provides the detail for the symbolic execution approach and its evaluation results. Chapter 6 discusses the proposed combinatorial approach and, respectively Chapter 7 presents how symbolic evaluation is leveraged to improve the weaknesses of combinatorial testing. Finally, Chapter 8 concludes this dissertation with the discussion of the contributions, limitations and the future work.

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

- Nariman Mirzaei, Joshua Garcia, Hamid Bagheri and Sam Malek. Reducing Combinatorics in GUI Testing of Android Apps. To appear in the 38th International Conference on Software Engineering (ICSE), Austin, TX, May 2016

- Nariman Mirzaei, Hamid Bagheri, Riyadh Mahmood and Sam Malek. "SIG-Droid: Automated System Input Generation for Android Applications". in Proceedings of the IEEE 26th International Symposium on Software Reliability Engineering (ISSRE), Gaithersburg MD, November 2015.

- Riyadh Mahmood, Nariman Mirzaei, Sam Malek. " EvoDroid: Segmented Evolutionary Testing of Android Apps". in Proceedings of the 2014 ACM SIG- SOFT International Symposium on Foundations of Software Engineering, ser. FSE 14. Hong Kong, China: ACM, November 2014.

- Nariman Mirzaei, Sam Malek, Corina S. Pasareanu, Naeem Esfahani, and Riyadh Mahmood. " Testing Android Apps Through Symbolic Execution". 2012 JPF-Workshop, Research Triangle Park, North Carolina, USA, November 2012.

- Riyadh Mahmood, Naeem Esfahani, Thabet Kacem, Nariman Mirzaei, Sam Malek, and Angelos Stavrou. "A Whitebox Approach for Automated Security Testing of Android Applications on the Cloud". The 7th International Workshop on Automation of Software Test (AST 2012), Zurich, Switzerland, June 2012.

Chapter 2: Background and Related Work

This chapter provides an overview of Android followed by a brief review of the existing literature on input generation techniques and Android testing.

2.1 Android Framework

The Google Android framework includes a full Linux operating system based on the ARM processor, system libraries, middleware, and a suite of pre-installed applications. It is based on the Dalvik Virtual Machine (DVM) [22] for executing programs written in Java. The Android ADF provides an API for application development and includes services for building GUI applications, data access, and other component types. The framework is designed to simplify the reuse and integration of components. Applications publish their capabilities and others can use them subject to certain constraints.

Android apps are built using a mandatory *AndroidManifest.xml* file. The manifest file values are bound to the application at compile time and cannot be changed afterwards unless the application is recompiled. This file provides essential information for managing the lifecycle of an application to the Android ADF. Examples of the kinds of information included in a manifest file are descriptions of the application's components among other architectural and configuration properties.

Components can be one of the following types: Activities, Services, Broadcast Receivers, or Content Providers. An Activity is a screen that is presented to the user and contains a set of layouts (e.g., LinearLayout that organizes items within the screen horizontally or vertically). The layouts contain GUI controls, known as view widgets (e.g., TextView for viewing text and EditText for text inputs). The layouts and their controls are typically described in a configuration XML file with each layout and control having a unique identifier.

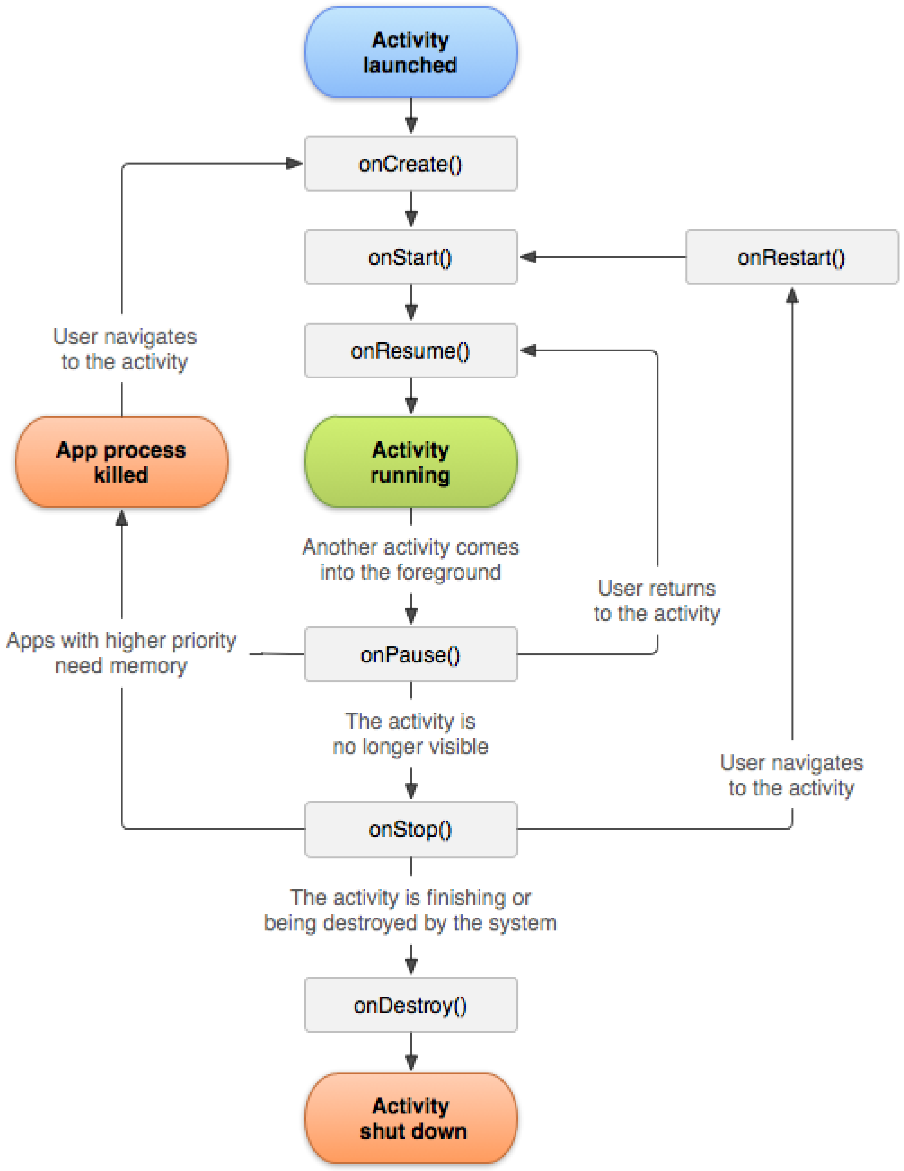
Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Figure 2.1: Android Activity Lifecycle [1].

A Service is a component that runs in the background and performs long-running tasks, such as playing music. Unlike an Activity, a Service does not present the user with a screen for interaction. A Content Provider manages structured data stored on the file system or database, such as contact information. Finally, a Broadcast Receiver responds to system-wide announcement messages (e.g., messages indicating the screen has turned off or the

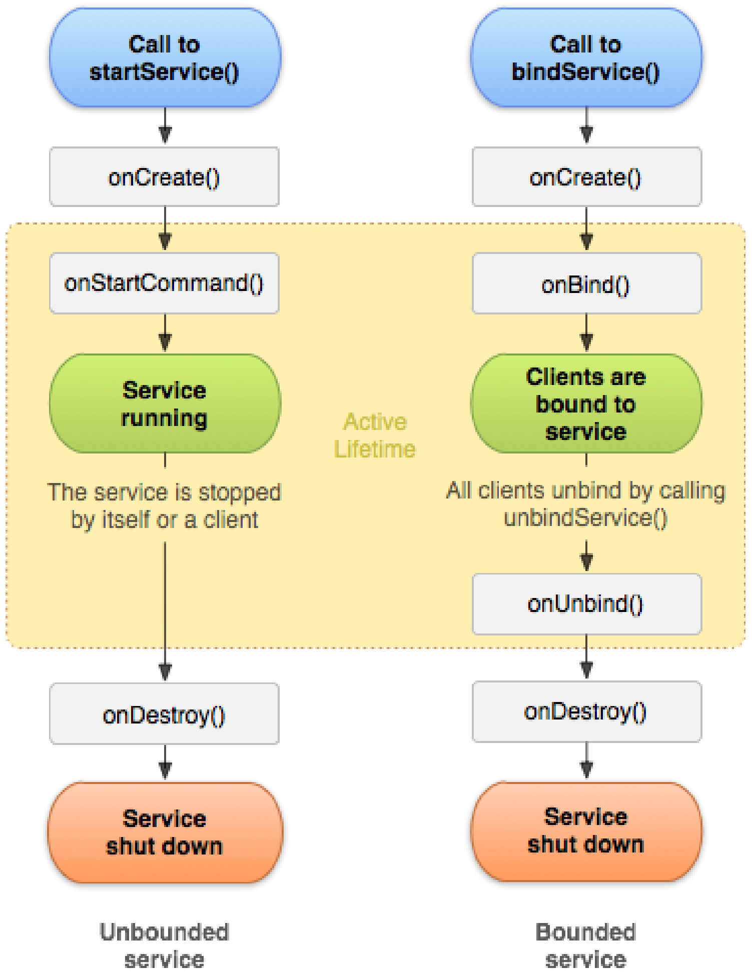

Figure 2.2: Android Service Lifecycle [2].

battery is low).

Activities, Services, and Broadcast Receivers are activated via Intent messages. An Intent message is an event for an action to be performed along with the data that supports that action. Intent messaging allows for late run-time binding between components, where the calls are not explicit in the code, rather connected through event messaging, a key property of event driven systems.

All major components, including Activities and Services, are required to follow pre-specified lifecycles [18] managed by the ADF. For instance, Figure 2.1 shows the events in the lifecycle of an Activity: `onCreate()`, `onStart()`, `onResume()`, `onPause()`, `onStop()`, `onRestart()`, and `onDestroy()`. Figure 2.2 presents the lifecycle of a service. These lifecycle event handlers are called by the ADF and play an important role in our research as explained later.

Android software development kit (SDK) provides developers with the ability to create Android Virtual Devices (AVD) (a.k.a emulators) [23] to simulate and imitate real devices. The AVD is essentially meant for testing Android applications on various Android versions and devices. Emulators mimic real Android devices and provide the user with *most* of the functionalities of a real device. The emulator is pre-loaded with a set of applications (as shown in the in Figure 2.3) like a browser, a phonebook and a map application, among other features.

The Android emulators are used extensively in this research to install the application under test as well as the test application that executes and instruments the application under test. The Android Debug Bridge (ADB) [24] tool is a command line tool that lets users communicate with an emulator or connected device.

2.2 Android Testing

There are several existing test input generation tools for Android with different primary goals such as detecting existing faults in Android apps, or maximizing the code coverage. Typically, it is assumed that the app developers are the main users of the tools. These tools assume that the source-code of the app is available and their main goal is to help the developer to improve the quality of their apps by catching possible faults and fixing them. The ability to generate tests has applications beyond testing for functional defects. Energy issues, latent malware, and portability problems are important concerns in the context of mobile devices that are often effectively detected by executing the code.

As Android apps are event-driven, i.e., the program is running in the idle state waiting for the user to interact with the app via the GUI, or some type of system events from the Android OS. In Android, GUI events include user-actions such as clicks, scrolls, or system events, such as the GPS location update. Users may also use the GUI to enter certain values for the widgets. These inputs can be entering a value in a text-box, selecting a certain item in a list, entering a specific value into a text-box and so on. Testing tools either treat all user actions as events or break them down into sequences of events that simulate the user actions and input values for the data-widgets on the GUI. Both the sequences of events and the inputs can be generated either randomly or by following a systematic approach. In the

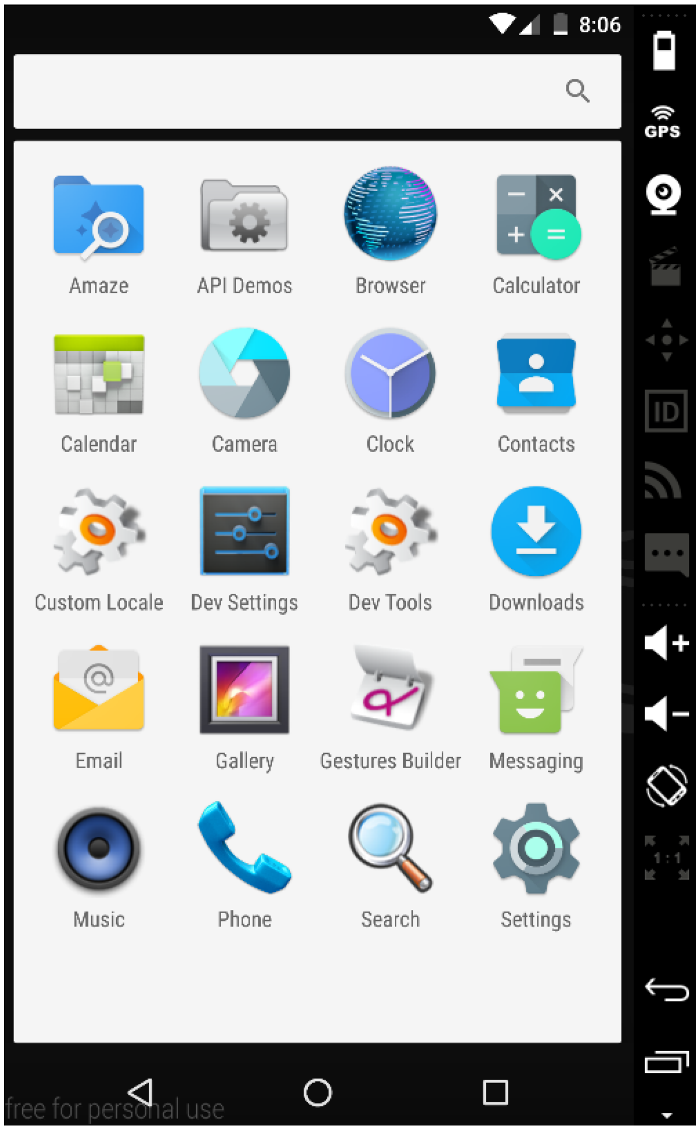

Figure 2.3: Android Emulator Screenshot.

latter case, generally a model of the app is used to guide the process and limit the search space. These models can be constructed statically, dynamically, or completely manually.

Current techniques for dealing with traditional event based systems either use capture-replay or model driven approaches. In capture-replay approaches [25], the user records her interaction sequences with the GUI, which are replayed at time of testing. Model driven techniques such as [26, 27] require the user to provide a model of the software system's usages. Both capture-replay and model driven approaches depend on manual effort, thus are not very convenient, and are prone to missing ways in which an app could be engaged and the human tester is unaware of. There have also been efforts to extract directed graph models automatically by crawling the GUI [28, 29], and use those graphs to generate test sequences, but again may fail to identify other ways a system can be engaged with.

2.2.1 Android Test Automation Frameworks

The Android development environment ships with a powerful testing framework, i.e., Android Testing Framework [30]. The Android Testing framework is built on top of JUnit and includes an API that extends the JUnit API with an instrumentation framework and Android specific testing classes. It also includes additional utility classes such as an extra set of Assertion classes for Android specific concepts (such as different Android Views), and a set of classes for testing various Android components of an application (Activity, Content Provider, and etc.).

Robotium [31] is an open-source testing framework for Android applications that is also built on the top of Junit framework. Robotium provides additional GUI assertions like in Web application testing using the Selenium framework. It is mostly used for Black-box testing and to assess applications at functional, system, and acceptance level.

Uiautomator [32] provides a set of APIs for the test engineers to build GUI tests that emulate user interactions such as clicking buttons, filling textboxes, and swiping. The Uiautomator testing framework is useful Black-box testing of apps through GUI. It provides support for GUI assertions and the ability of checking the state of apps GUI at before and after each action.

Monkey-Runner [33] provides an API for writing tests that control an Android device or emulator from outside of Android code. Monkey-Runner captures the results as screenshots and can be used for regression testing by comparing them as well as functional testing.

Esspreso [34] is the latest Android test automation framework by Google. It builds on top of existing instrumentation infrastructure that Monkey-Runner provides an API to interact with the UI more reliably.

Finally, Robolectric [35] is a testing framework that separates the test cases from the device or emulator and provides the ability to run the tests directly by referencing the Android library files. It provides lightweight shadow objects for Android framework library classes, that mimic the actual classes behavior. It replaces the body of Android API methods at run-time using java reflection. These shadow classes can be used for the purpose of unit testing outside an emulator or a real device.

While all above frameworks automate the execution of the tests, they do not provide a way for generating the test cases automatically, hence the test cases themselves still have to be manually developed. In contrast the main focus of this dissertation is on building techniques for generating test inputs automatically.

2.2.2 Random and Pseudo Random Techniques

The state of the practice in automated testing of Android apps is random testing. In fact, a recent study of existing tools by Choudhary et al. [36] claims Android Monkey [37], a random-testing program for Android, to be the winner among the existing test input generation tools.

Android events consist of GUI events that are initiated by the user and system events triggered by the Android framework itself. Usually, Android applications are registered to handle only a handful of system events, and only under specific conditions. As a result, testing system events randomly is quite inefficient and most of the random testing techniques for Android (such as [8,37]), focus on generating only GUI events. Another group of random testing techniques (such as [38]) focuses on testing inter-application communications by randomly generating values for Intents (a.k.a Intent fuzzing). Intent fuzzers mainly aim to test the robustness of the apps and reveal security vulnerabilities by generating invalid and malicious intents.

Android Monkey [37] is a brute-force mechanism that often achieves shallow code coverage. It is part of the Android developers toolkit and is widely used by both developers and app market managers. Monkey utilizes a completely random strategy, and randomly fires off both GUI and system events based on the number of events that are specified by the tester.

Dynodroid [8] also uses random values and sequences of events, but it incorporates several heuristics to improve on Android Monkey's performance. It generates only system events that are relevant for the application by checking the *android-manifest.xml*. It also tracks the type and number of events used so far and uses a least recently used algorithm to pick the next event. Finally, Dynodroid provides the tester with the ability to manually enter values for specific types of inputs (e.g. text-boxes).

Several other approaches build on random testing techniques. Amalfitano et al. [9] describe a GUI crawling-based approach that leverages completely random inputs to generate unique test cases. Hu and Neamtiu [6] present a random approach for generating GUI tests that uses the Android Monkey to execute.

Although, these random testing techniques can generate events efficiently, they are not suitable for generating highly specific inputs. Moreover, they are do not keep track of part of the application that has been already covered, and are likely to generate redundant events. In contrast the major focus of this dissertation is on generating intelligent test inputs systematically with the goal of maximizing the statement coverage.

2.2.3 Model Based Techniques

Following the path of previous testing techniques for event-based system such as web-based applications, some Android testing tools extract a GUI model of the application to systematically generate sequences of events that resemble the behavior of the application. These tools employ static and dynamic analysis techniques and generate a finite-state machine that captures the activities of an app as the states and the events as the transitions.

MobiGUITAR [39], built on the top of famous GUITAR [40] framework uses GUI ripping to build a model of an app. The model then is traversed by a depth-first search strategy to generate test cases. The exploration is restarted from the starting state when the tool cannot detect new states during the exploration. MobiGUITAR can use either random or predefined constant input values during the exploration.

ORBIT [7] is a grey-box model extraction technique that creates a GUI model of the app for testing. While *ORBIT* implements the same exploration strategy as MobiGUITAR it uses static analysis to identify relevant UI events for a specific activity as the transitions between the states.

SwiftHand [5] is a GUI testing technique that uses dynamic analysis and machine learning to infer a finite state model of the app during testing. The inferred model is used to generate UI events that visit unexplored states of the app. The model is refined dynamically during the execution of the app using the generated inputs. The main focus of SwiftHand is to optimize the exploration strategy in order to minimize the restarts of the app during the exploration.

A^3E [41] is a static taint analysis technique for building an app model for automated exploration of an app's Activities. While the main focus of A^3E is on the construction of models for testing, rather than the generation of GUI tests in a systematic approach, two exploration strategies are used to test the app: a depth first search strategy and a targeted based strategy with a goal of reaching a certain state in the model.

PUMA [42] is built on the top of Uiautomator [32] and provides the infrastructure for dynamic analysis of apps. PUMA is equipped with Monkey's exploration strategy but it provides a framework that can be extended to implement any exploration strategies. Moreover, it provides an environment for analysis and modification of the finite-state model of the app at run-time.

Unlike the work presented in the dissertation, these approaches focus on the construction of models for testing that are covered using a depth-first search strategy for generation of event sequences and random input data. This work differs from them as I use prime path coverage, which subsumes all other graph coverage criteria, to generate the event sequences. I further generate the inputs for GUI widgets systematically through symbolic execution and combinatorial testing rather than using randomly generated input.

2.2.4 Record and Replay Techniques

MonkeyRecorder [43] and RERAN [44] implement record and replay techniques for Android apps. MonkeyRecorder allows testers to record a script for GUI events of an application on the device and run it. MonkeyRecorder only collects click, swipe, and text-input events.

RERAN logs the event system commands of the Android operating system to generate low-level event traces. These scripts are analyzed and turned into runnable scripts. RERAN replays the recorded script and it does not use any recombination of the recorded events for replay; moreover, the recorded scripts are hardware-specific including events coupled to screen locations. As a result, RERAN scripts cannot be easily ported to be used for other devices.

Although record and replay techniques can be useful for stress testing and regression testing, the scripts are generated manually and as a result they are usually biased towards only certain features and do not capture the behavior of the app completely. These techniques can only replay what is recorded and do not consider other combinations of events for replay. Moreover, the recorded scripts are often specific to a certain device and cannot be used for other devices (e.g., recorded script on a Nexus 6P will not work on a Galaxy S6). Unlike these works, the presented techniques in this dissertation provide a solution for both generating the sequences of events and input values for GUI widgets automatically.

Additionally, these test inputs can be used on any Android device that uses the Android API the test inputs are generated for.

2.2.5 Other techniques

Some application behavior can only be revealed upon providing specific inputs. This is the reason why some Android testing tools use more sophisticated techniques such as symbolic execution and evolutionary algorithms to guide the exploration towards previously uncovered code.

Jensen et al. [45] presented a system testing approach that combines symbolic execution with sequence generation. The main goal of this work is to find valid sequences and inputs to reach pre-specified target locations, and not maximizing the code coverage.

ACTEve [46] is an approach based on concolic testing of a particular Android library to identify the valid GUI events using the pixel coordinates by instrumenting both the framework and the app under test. While ACTEve handles both system and UI events, the proposed approach only focuses on testing screen tap events and does not address the problem of handling user's input values.

EvoDroid [47] employs evolutionary search that is guided by two models of the system to generate relevant sequences of events and inputs for apps. EvoDroid breaks down the app into smaller segments where each segment is essentially an Android component and is represented by an individual. The fitness function evaluates the quality of each test-case and the individuals are mutated accordingly to maximize the coverage.

These techniques focus on the problem of generating complex sequences of events through program analysis or evolutionary testing. On the other hand, this research mainly targets a complimentary problem, i.e., generating proper input values for GUI testing of Android applications systematically.

2.3 Test Input Generation

The amount of research on automatic test case generation in the past decades has shown its strong impact on the effectiveness and efficiency of whole testing process [48–50]. As a result, a good number of different techniques of test case generation has been advanced and investigated intensively.

This section provides a brief overview of the related literature on three well-known input generation techniques that are used in this research.

2.3.1 Model-based Input Generation

Model based input generation is an approach in which a model of the system under test is used as a reference to generate the test cases [50]. The model provides an abstraction of the behavior of the system and describes possible sequences of transitions between its states based on the app implementation. Model-based techniques can be divided into three categories: axiomatic approaches, finite state machine (FSM) approaches, and labeled transition system (LTS) approaches. [4, 51]

Finite state machines are frequently used in the domain of GUI testing in which a node of the FSM represents a state of the application and the state is identified by the values of graphical objects [19,52–54]. A transition of the FSM represents an application event/action (e.g., a method call, an event handler) that can change the application state, if executed. Additionally, guards and conditions can enrich the model to capture the context in which events and actions are executed.

The test cases are generated using a model exploration technique and a coverage criterion [55]. For example, in node coverage criterion every FSM state needs to be exercised by at least one test case, where edge coverage criterion ensures that every transition is executed [56]. These event sequences represent a usage scenario of the system under test in a test case.

All of these techniques focus on using the GUI model to generate sequences of events.

Thus, they lack a systematic way of generating user input values for data widgets. As a result, most MBT techniques use random or manually generated inputs. This can result in lower code coverage, particularly if the program specifies constraints on user inputs as well as missing potential faults. The research presented in this dissertation builds on an MBT approach and complements it with techniques that can generate proper inputs values for GUI widgets systematically.

2.3.2 Symbolic Execution

In 1975 King [13] introduced symbolic execution, a program analysis technique in which symbolic values are used as program inputs instead of concrete values. Consequently, the outputs of the program are transformed to a function of the symbolic inputs. The path condition is a Boolean formula over the symbolic values representing the constraints that must be satisfied in order for an execution to follow a specific path. Using the path conditions around symbolic values, a decision tree, called symbolic execution tree, is created.

For illustration of this technique, consider the Java program depicted in Figure 2.4a, where S0, S1, S2, S3, and S4 denote statements that can be invoked in different paths of the program. Clearly, random testing is not likely to result in good coverage for this program. Consider that the input value for y has to be exactly three times the value of variable x to cover statement S0. This is precisely where the symbolic execution is shown to be fruitful. Figure 2.4 shows the symbolic execution tree for this program. With the help of an off-the-shelf SAT solver, actual input values that result in paths shown in Figure 2.4b can be generated. These inputs can be used to generate test cases that cover different paths.

As an example, let X and Y be the symbolic representation of variables x and y, respectively. By solving the following constraint "$X > 0$ & $X \leq 3$ & $Y = X \times 3$", we obtain two values "$X = 3$" and "$Y = 9$", which result in taking the bold path in Figure 2.4 and executing S0 and S3. Similarly, using symbolic execution, we can generate all possible inputs for the test method in such a way that all feasible paths in the program are explored. Moreover, symbolic execution can determine infeasible or unreachable paths and report an

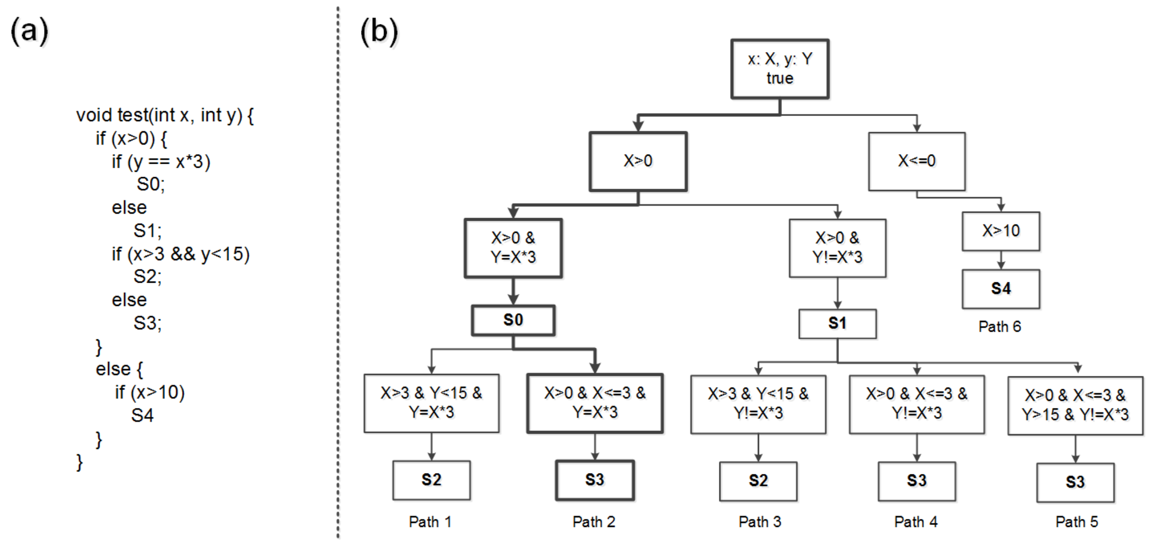
```
void test(int x, int y) {
  if (x>0) {
    if (y == x*3)
      S0;
    else
      S1;
    if (x>3 && y<15)
      S2;
    else
      S3;
  }
  else {
    if (x>10)
      S4
  }
}
```

Figure 2.4: Symbolic execution: (a) sample code, and (b) the corresponding symbolic execution tree, where X and Y are the symbolic representations of variables x and y

assertion violation (path 6).

Recently, there has been an increasing attention towards symbolic execution techniques. This is mainly a result of the inception of new powerful constraint solvers (e.g. Z3 [57] and Yices [58]), which can solve complex and large constraints, and the low prices and accessibility of computational power. Since its inception, symbolic execution has been mostly used to generate test data with goals such as to improve code coverage and expose software faults [59–61].

Symbolic execution is computationally expensive and it is difficult to symbolically reason about of all paths of significantly large programs, as most real world software have an extremely large number of paths. Many techniques have been proposed to alleviate the path explosion problem, among those dynamic symbolic execution (a.k.a concolic execution) stands out.

Dynamic Symbolic Execution first executes the program normally along the path with some random inputs [62, 63]. Next, the path constraints for each path the program takes are computed. Finally, the path is also symbolically executed to compute new inputs that drive the program along alternative paths.

Pasareanu et al. [64] proposed using concrete values to simplify complex constraints. Unlike previous Concolic Execution techniques, concrete values are not used to obtain the path conditions from normal execution. The use of concrete values is limited to only complex parts of the code or the external API calls that are not needed to be symbolically executed to simplify the complex constraint.

A number of tools for symbolic execution are publicly available. Symbolic Pathfinder (SPF), [65] is the best-known symbolic execution engine for Java programs. It is built on top of Java Pathfinder (JPF) [17], an open source general-purpose model checker for Java programs. Unlike other symbolic execution engines, SPF does not work with code instrumentation. It works with a non-standard interpretation of Java byte-code using a modified JVM [17]. SPF analyzes Java byte-code and handles mixed integer and real constraints, as well as complex mathematical constraints through heuristic solving. SPF can be used for test input generation and finding counterexamples to safety properties [65]. Other well-known available symbolic execution tools include JCUTE [66] and JFuzz [67] which target Java, CUTE [68], and Klee [69] which target C and, Pex [70] for .NET languages.

Symbolic execution has been used for testing graphical interfaces. Barad [14] symbolically executes a sequence of events to infer inputs for data components of the GUI and relies on a random technique for generating the sequences of events. More recently, Jensen et al. [45] proposed an approach that combines symbolic execution with sequence generation. Their work is mainly concerned with finding valid sequences and inputs to reach pre-specified target locations.

Among well-known input generation techniques, symbolic execution is quite unique due to its uses of program analysis and constraint solvers. However, it has been shown that symbolic execution by itself could be ineffective for reasoning about event sequences and is best when it is used in combination with other techniques such as evolutionary testing [16].

Among prior works on symbolic execution only a few have focused on using symbolic execution to generate test inputs for GUI applications (e.g. [14, 45, 46]) but non of them targets the problem generating system-level test inputs. Not only this dissertation presents the first symbolic execution engine for Android applications, but also it introduces SIG-Droid, a framework for generating system test inputs through symbolic execution.

2.3.3 Combinatorial Testing

Combinatorial testing [20] is a brute-force approach that tries all possible combinations on inputs given a set of input parameters. For a system with p parameters, where each parameter can take v possible values, the number of all possible combinations is p^v. Hence, exhaustive combinatiorial testing is often computationally prohibitive. *T-way* combinatorial testing is a heuristic-based alternative approach to exhaustive testing [71] where all combinations of any $t \leq p$ parameters have to be covered. The most common type of t-way testing is pair-wise testing [15].

In combinatorial testing literature, a large number of techniques employ heuristics to generate minimal set of test combinations. Pair-wise testing has shown to be an effective heuristic for reducing the number of combinatorial tests [12, 72]. Approaches such as [73, 74] propose using greedy or heuristic algorithms to generate minimal sets of tests for a given combinatorial criteria. Some newer algorithms, use meta-heuristic search techniques (such as genetic algorithms) to optimize selection samples [75]. However, the problem of generating minimal sets of test cases that satisfy a given combinatorial criteria is NP complete and the optimal solution cannot be obtained in non-trivial cases [76].

Combinatorial testing has shown to be an effective approach in GUI-based testing [77]. Nguyen et al. [19] proposed an approach that leverages manually constructed behavioral models of an app in pairwise testing of GUI-based applications. While using models that are generated manually by the engineers has some advantageous (such as more complete and accurate models), not automating model generation severely restricts the applicability of this approach.

Kim et al. [78] introduced the idea of using static analysis to filter irrelevant features when testing software product lines. Petke et al. [79] showed that higher strength of t-way testing can be practical and more effective in the presence of constraints. Other areas of interest for combinatorial testing include data-intensive system, system configurations and software product lines (e.g., [78, 80, 81]).

This work differs from these approaches as it (1) specifically targets Android apps, (2) automatically extracts the models through program analysis, (3) uses prime paths to generate the sequences of events, (4) relies on a number of heuristics to determine the interacting widgets in order to reduce the number of tests without degrading the coverage, and (5) uses program analysis to determine the best values to represent the input domain partitions of each widget.

Chapter 3: Research Problem

With well over a million apps [82], Android has become one of the dominant mobile platforms [3]. Android app markets, such as Google Play, have created a fundamental shift in the way software is delivered to consumers, with thousands of apps added and updated on a daily basis. The majority of these apps are developed at a nominal cost by entrepreneurs who do not have the resources for properly testing their software. Hence, there is an increasing demand for applicable automated testing techniques. One key obstacle towards test automation for Android apps that are heavily GUI-driven is the lack of practical techniques for generation of test cases.

While there has been tremendous progress in Android testing at both unit-level and GUI testing, random testing still remains the major player in automated system testing of Android apps [36]. Monkey [37], a popular fuzzing tool provided by Google generates random touchscreen presses, gestures, and other system-level inputs. Dynodroid [8] performs more effective event-aware random testing, through inferring representative set of events and employing certain heuristics. Other techniques [5–7, 9, 46] mainly focus on testing the program through GUI elements. System behaviors dependent on data values, though, have not been adequately considered to a large extent, as data widgets are abstracted away, which may cause shallow code coverage.

System testing of interactive applications, such as Android apps, can be broken down into two distinct, yet interwoven problems. The first is to generate sequences of unique events, where each sequence represents a particular app use-case and causes a change in the state of the app. Here, the whole set of sequences exhaustively cover all possible use scenarios. The second is to generate proper values for GUI data widgets that take user inputs, such as textboxes. Here, the input domain can be quite large.

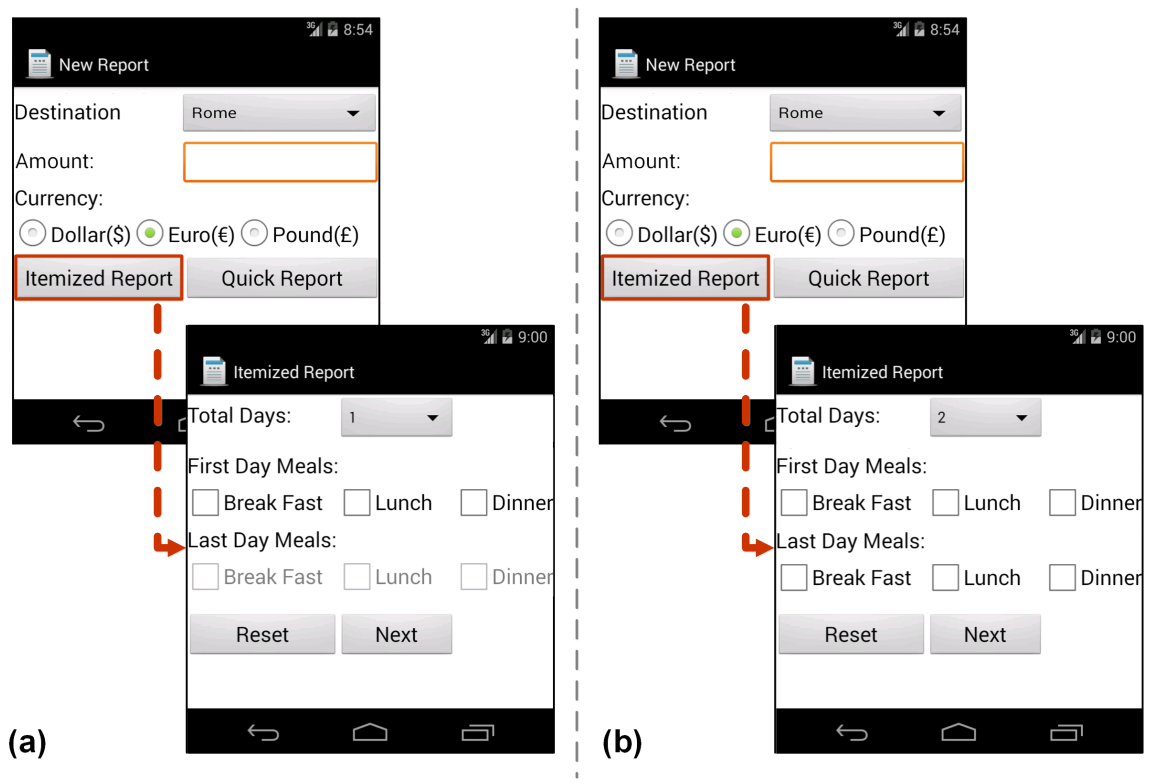

Figure 3.1: Screenshots for a part of ERS app: (a) when total days is 1, the check-boxes for *Last Day Meals* are disabled, and (b) when the total days is greater than 1, the check-boxes for *Last Day Meals* are enabled.

The reminder of this chapter uses an example to describe the details of the research problem. This is followed by presenting the problem statement, and enumerating the research hypotheses.

3.1 Motivating Example

For illustrating the proposed approaches, a simple Android app, Expense Reporting System (ERS), is used in this document. The ERS app allows a user to submit meal expenses incurred during a trip on an Android device. Figure 3.1 depicts two of ERS's *Activities*: *NewReportActivity* and *ItemizedReportActivity*.

The *NewReportActivity* is the *main* Activity, i.e., it is the first screen presented to the user when an app is invoked. From the *NewReportActivity*, the user can select the

Destination, enter an allowable expense *Amount*, the *Currency*, and initiate the creation of two types of reports: *Itemized Report* and *Quick Report*. The *ItemizedReportActivity* allows the user to enter an itemized list of meal expenses, including (1) the total days of the trip, and (2) the number of meals purchased on the trip's first and last day. When *Total Days* is 1 (i.e., the first and last days are the same), the check-boxes corresponding to the last day meals are disabled (see Figure 3.1a). On the other hand, the *QuickReportActivity* (not shown in Figure 3.1 for brevity) allows the user to provide an aggregate number for the meal expenses incurred on a trip. As explained in Chapter 2, the widgets on each activity should be defined in an XML layout file. Listing 3.1 presents a snippet of the layout file for `NewReportActivity`.

Listing 3.2 shows code snippets realizing one of the functionalities provided by this app. When the *"Quick Report"* button (see Figure 3.1) is clicked, the `onClick` method is called by the ADF. Subsequently, if the amount is less than \$500, an Intent is sent to the `QuickReportActivity` including as payload the amount, currency and destination accounts.

Recall that testing of Android apps, such as ERS, can be broken down into problems of generating sequences of unique events and augmenting those sequences with values for GUI data widgets. Model-based testing helps us to understand and test the event-based nature of a system under test [10, 11]. In other words, we can capture the GUI screens of an app and the events that result into transitions between the screens in the form of a finite state machine (*FSM*). This *FSM* can be used to generate sequences of unique events. Each path in the *FSM* can potentially result into a viable test-case. For example the path $\{a_0 \xrightarrow{onClick(QuickReport)} a_1\}$, is a conceivable sequence for a test case that brings up the screen corresponding to `NewReportActivity`, and clicks on *"Quick Report"* button. But this test case needs to also include values for the data widgets on `NewReportActivity`, i.e., amount, currency, and destination.

Existing model-based approaches (e.g., [11]) take the paths in the model and complete them with either manually-defined or random values. Here, the main problem is that these approaches neither consider: (1) the possible constraints on the values that correspond to

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"
   >
    ...
    <TextView
        android:id="@+id/amountLbl"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Amount"
        android:textAppearance="?android:attr/textAppearanceMedium" />
    <EditText
        android:id="@+id/amountId"
        android:layout_width="match_parent"
        android:layout_height="wrap_content">
      <requestFocus />
    </EditText>
    <Button
        android:id="@+id/quickReportBtn"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text=Quick Report />
    ...
</LinearLayout>
```

Listing 3.1: snippet from QuickReport.xml layout app.

individual widgets, nor (2) the interactions among data input widgets. Thus, they might result in shallow coverage and miss faults that are result of interactions among some input values. For example, in the case of ERS, consider the constraint in the code that the amount cannot be more than $500 for quick reports. In current model-based approaches, there is no systematic way of generating tests that cover both possible paths following the constraint.

Symbolic execution and combinatorial testing are two input generation techniques that have shown to be quite effective for generating values that properly represent the input domain of a system under test [13–15]. Symbolic Execution [13] is a well-known program analysis technique that can be used to generate appropriate values for constrained inputs to

```
public class NewReportActivity extends Activity {
  ...
  public class QuickReportButton implements OnClickListener {
    public void onClick (View v) {
      ...
      EditText amount = (EditText)findViewById(R.id.amountId);
      amountValue = Integer.parseInt(amount.getText().toString());
      if (amountValue <= 500) {
        ...
        Intent intent = new Intent(this, QuickReportActivity.class);
        intent.putExtra("amount", amountValue);
        intent.putExtra("currency", currency);
        intent.putExtra("destination", destination);
        startActivity(intent);
      } else {
          throw new AmountException(Limit for Quick Report Exceeded)
      }
    }
  }
  ...
}

public class QuickReportActivity extends Activity {
  ...
  public void onCreate (Bundle savedInstanceState) {
    ...
      Intent intent = getIntent();
      int amount = intent.getExtra("amount");
    ...
    }
    ...
}
```

Listing 3.2: Code snippets from NewReportActivity and QuickReportActivity app.

explore all possible execution paths in a program. For example, by using symbolic execution for the path $\{a_0 \xrightarrow{onClick(QuickReport)} a_1\}$ in ERS, we can generate two values of \$501 and \$1 for amount textbox.

Although symbolic execution itself is well understood and performing symbolic execution on simple programs is straightforward, efficient and automatic symbolic execution of real-world programs at the system-level is usually not achievable [4]. As a result symbolic execution is mainly used at unit level. Moreover, symbolic execution cannot efficiently deal complex constraints that involve interaction of input parameters (which would result into complex path conditions).

On the other hand, combinatorial testing techniques deal with the problem of exercising relevant combinations of values for input parameters of a program [15]. *Exhaustive* variation of combinatorial testing is often viewed as impratical due to the explosion of possible combinations [15]. A more practical alternative is *t-way* combinatorial testing [15], where all combinations for only a subset of input parameters (i.e., t) are considered [20]. But even under t-way testing, the number of generated test cases could grow rapidly. Moreover, without a systematic approach to determine the dependencies, arbitrary selection of t widgets to be combinatorily tested is bound to be less effective than an exhaustive approach in terms of both code coverage and fault detection.

Although combinatorial testing has shown to be promising for testing GUI application its success depends on how well the input domain of widgets are partitioned. For example, in ERS app if the input domain for `amount` is manually partitioned into three classes of $\{1000,\ 10000,\ 100000\}$, the value combinations will only exercise one path and ignore the other. Here, a program analysis technique such as symbolic execution in a relatively small scale, i.e., an activity, can be used to systematically generate values that represent the input classes for unbounded data-widgets in Android apps.

3.2 Problem Statement

The problem caused by lack of automated techniques to effectively test Android applications can be summarized as follow:

> *There has been an explosive growth in the number of new apps for mobile platforms, such as Android. Due to poor quality of the submitted apps, we are witnessing a high ratio of app removal from the app markets. Not surprisingly, there is an increasing demand by developers, consumers, and market operators for practical techniques to generate test inputs that can rapidly assess and test the robustness of apps submitted to online app markets.*

3.3 Research Hypotheses

This research investigates the following hypotheses:

- While model based testing is shown to be promising for generating sequences of actions in event-driven environments such as Android, it lacks the ability to systematically generate the input data required by such sequences. The input domain of proper values for GUI data widgets can be quite large. As a result random values can be quite ineffective. For example, in a numeric textbox that accepts 5 unsigned digits and involves a conditional statement satisfied when the input value equals a certain integer, random input generation has only $\frac{1}{5^{10}}$ chance to reach the state satisfying that condition. Symbolic execution [13] is a program analysis technique that can be used to systematically generate values for exploring all the paths in a program.

  > **Hypothesis 1:** *Symbolic execution can be used in combination with a model-based approach to generate system-level test-cases for Android application.*

- An opportunity to automate the testing activities in Android is presented by the fact that apps are developed on top of an Application Development Framework (ADF).

The Android ADF ensures apps developed by a variety of suppliers can inter-operate and coexist together in a single system (a phone), as long as they conform to the rules and constraints imposed by the framework. The Android ADF constrains the lifecycle of components comprising an app, the styles of communication among its software components, and the way in which GUI widgets (e.g., buttons, check-boxes) and other commonly needed functionalities (e.g., GPS, camera) can be accessed. An underlying insight in this research is that the knowledge of these constraints along with the metadata associated with each app can be used to automate many software testing activities, specifically combinatorial testing of application.

> **Hypothesis 2:** *By using only widgets and activities that have dependencies on one another as candidates for t-way combinatorial testing, an efficient alternative approach for exhaustive combinatorial testing that achieves comparable code coverage can be developed.*

- Combinatorial testing is shown to be effective for testing various combinations of parameters [15]. Commonly, predefined sets of input classes are used for testing unbounded parameters. The number of input classes has an exponential impact on the number of final test inputs. By determining the input classes more systematically using program analysis and constraint solving, a fewer number of inputs can achieve identical or even improved code coverage.

> **Hypothesis 3:** *A program analysis technique can be devised to determine the input classes for unbounded data-widgets in combinatorial testing of Android apps to improve the effectiveness of test inputs in terms of code coverage.*

Chapter 4: Model Extraction

As mentioned in Chapter 1, testing of interactive applications, such as Android apps, is comprised of two parts: generating sequence of events (e.g., button clicks) and selection of input values (e.g., drop-down menu choices) [14]. A model-based approach can be used to generate sequences of unique events, where each sequence represents a particular app use and causes a change in the state of the app [47, 83, 84]. Here, the whole set of sequences exhaustively cover all possible use scenarios.

Android apps are built using a common application development framework (ADF) that ensures apps developed by a wide variety of suppliers can interoperate and coexist together in a single system (e.g., a phone) as long as they conform to the rules and constraints imposed by the framework. An ADF exposes well-defined extension points for building the application-specific logic, setting it apart from traditional desktop software that is often implemented as a monolithic independent piece of code. Android also provides a container to manage the lifecycle of components comprising an app and facilitates the communication among them. As a result, unlike a traditional monolithic software system, an Android app consists of code snippets that engage one another using the ADFs sophisticated event delivery facilities. This poses a challenge to test automation, as the app's control flow frequently interleaves with the ADF. On the other hand, the knowledge of ADF along with the metadata associated with each app can be used to automate many of the software testing activities, in particular test input generation, as illustrated in this document.

Using the specification of the app and Adroid ADF, *Model Extraction* extracts two types of models for each app: *Interface Model (IM)* and *Activity Transition Model (ATM).* In the remainder of this chapter, each model and its extraction process is described in details. As it is shown in Chapters 5 and 6, these model are used to generate sequences of events that represent possible use cases for the system can be produced by exploring the ATM.

4.1 Interface Model

The *IM* provides information about all of the GUI inputs of an app, such as the widgets and input fields belonging to an Activity. More formally, the IM is defined as follows:

Definition 1. The *IM* of an app is a tuple $\langle A, E, W, I \rangle$, where

- A is a finite, non-empty set of Activities of the app.
- E is a finite set of event handlers of the app (e.g., *onClick()* is the handler for a button click). Each Activity $a \in A$ has a set of event handlers $eHandlers(a) \subseteq E$.
- W is a finite set of GUI widgets of the app (e.g., a check-box, radio-button, drop-down menu, and etc.). Each Activity a has a set of widgets $widgets(a) \subseteq W$.
- I is a finite set of input classes for widgets of the app. Each widget w has a set of input classes $ic(w) \subseteq I$. Each input class is a partition of the input domain of each widget. For instance, input classes of a check-box are *checked* and *unchecked*, while input classes of a drop-down menu are its choices.

Model Extraction obtains the *IM* by analyzing the information contained in the meta-data included in an Android APK file, namely its XML-based *manifest* and *layout* files. More specifically, *Model Extraction* determines all the Activities comprising an app from its manifest file. Subsequently, for each Activity, *Model Extraction* identifies the corresponding layout file[1]. It then parses the layout file (e.g. Listing 3.1) to obtain all information for each widget, such as its name, id, input type, etc. Our current implementation extracts the input classes for widgets that provide users with a list of options, such as check-boxes and drop-down menus, directly from the layout files. *Model Extraction* uses the same layout files to divide the domain space of text-boxes into different classes based on the limits imposed on the text box values (e.g., max length). For unbounded text boxes, and other unbounded widgets, *Model Extraction* uses a configurable set of input classes that can be defined by the user.

[1]The corresponding layout file of an activity is set by the `setContentView()` method of the class that defines the activity [18].

As will be explained in Section 6.4, the IM is used to determine the GUI widgets and their properties as well as events comprising each activity for the generation of test cases.

4.2 Activity Transition Model

An *ATM* represents the high-level behavior of an app's GUI in terms of its Activities and the transitions resulting from invocations of its event handlers. More formally, the *ATM* is defined as follows:

Definition 2. The *ATM* of an app is a finite state machine represented as a tuple $\langle A, a_0, E, F \rangle$, where

- A is a finite, non-empty set of Activities.
- a_0 is the starting Activity (i.e., *main* Activity), defined in an app's manifest file.
- E is a finite set of directed transitions from the starting Activity to final Activities, labeled by event-handler names. Each transition represents an event handler and denoted as $a_i \xrightarrow{e_k} a_j$, where $a_i, a_j \in A$ and e_k is an event handler.
- F is a finite, non-empty set of final Activities in the *ATM*.

Figure 4.1 shows the ATM for the ERS app. To obtain an *ATM* such as this, *Model Extraction* first determines the Activities $A = \{a_0, a_1, a_2, a_3, a_4\}$ comprising the app from its manifest file. To determine the transitions between the Activities, *Model Extraction* first generates the call-graph of the app using Soot. It then performs a depth-first traversal of *main*[2] Activity's call-graph starting from its *onCreate()* method, which we know from Android's ADF specification to be the starting point of all apps. In the context of ERS, this corresponds to *NewReportActivity*'s *onCreate()* method. For each encountered node in the call-graph, *Model Extraction* checks whether it would result in an activity transition, and if so, adds it to set E. This is done by identifying the nodes where implicit method[3] calls

[2]If there are more than one activity defined as the main activity in app's manifest file, only the first one is considered.

[3]Implicit calls are method calls that are handle via Intent messaging system of Android ADF. Hence, they do not appear in the static call-graph generated by Soot.

Algorithm 1: Implicit Call Extraction

```
Input: $CG$ : set of activity call graphs, $\psi$ : set of implicit callers
Output: $\Upsilon$ : implicit calls
1 foreach $c \in CG$ do
2     $rootNodes \leftarrow c.getRoot()$
3 foreach $c \in CG$ do
4     $lNodes \leftarrow c.getImplicitCalls()$
5     foreach $l \in lNodes$ do
6         if $l \in \psi$ then
7             $d \leftarrow l.getDestination()$
8             if $d \in rootNodes$ then
9                 $\Upsilon.Add(l, d)$
```

are initiated. These nodes would have to be method calls that either set an event handler, start other activities, send Intent messages, or handle system events. System event handlers deal with notification events, such as when a call is received, network is disconnected, or the battery is running low.

Based on Android's specification, we know that the links would have to be from leaf nodes of the call-graph to other nodes. For example, in ERS there is an implicit call from `startActivity` in *NewReportActivity* to *ItemizedReportActivity*'s `onCreate()`. Algorithm 9 shows how implicit calls are extracted, given a set of call-graphs each of which belongs to an activity and the set of caller methods that initiate implicit calls. These methods are defined by Android's specification. As new call-graphs are linked and connected, they are traversed in a similar fashion. By doing so, *Model Extraction* is able to connect the entire call-graph of the application, from beginning to end. The call -graph model is updated with the newly found information.

A call may result in a transition in two ways:

1. *Inter-component transition:* these are implicit calls that result in the transfer of control from one Activity to another Activity. For instance, in the example of ERS in Figure 3.1,

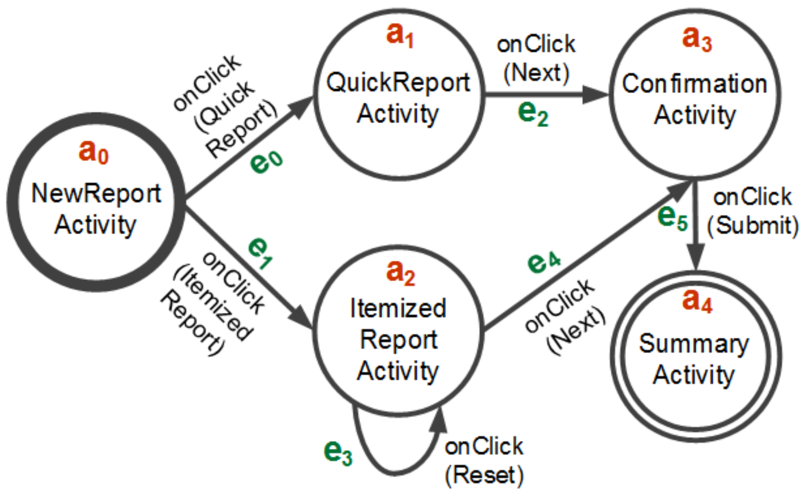

Figure 4.1: Activity Transition Model for the ERS app

when the *Itemized Report* button is clicked, the corresponding handler calls Android's *startActivity* method, which sends an Intent message resulting in the transfer of control to *ItemizedReportActivity*'s *onCreate()* method. In this case, *Model Extraction* extracts the destination from the Intent, and add the following transition $E = E \cup \{a_0 \xrightarrow{onClick(ItemizedReport)} a_2\}$.

2. *Intra-component transition:* these are implicit calls to GUI event handlers in an Activity that result in a transition back to the same Activity. For instance, the *ItemizedReportActivity* has a *Click* event associated with its *Reset* button. This event is handled by the Activity's *onClick()* method that is registered with that button. In this case, *Model Extraction* adds the following transition to the model: $E = E \cup \{a_2 \xrightarrow{onClick(Reset)} a_2\}$.

Upon traversing the call-graph of a_0, the above process repeats for all of the Activities remaining in A. Finally, *Model Extraction* populate the set F with the Activities that do not have any outgoing inter-component transitions, and if they do, it is only to nodes that precede them.

The *Model Extraction* component is implemented on top of Soot, a static-analysis framework for Java [85]. To analyze an Android app, it utilizes the Dexpler transformer [86] to translate Android's Dalvik bytecode to Jimple, Soot's intermediate representation. By leveraging Soot and Dexpler, this approach works with an app's source code as well as its APK file. The model generator can handle most listeners for common widgets such as `onClickListener`, `onLongClickListener`,`onOptionsItemSelected`, and so on.

Chapter 5: Input Generation for Android Applications Using Symbolic Execution

Symbolic execution is a promising automated testing technique that can effectively deal with constraints. While symbolic execution has proven to be effective for unit level testing, our goal is to utilize symbolic execution for end-to-end system testing of Android apps.

Symbolic execution of programs that are developed on the top of an ADF, however, has always been challenging due to problems such as *path-divergence* that occurs when a symbolic value flows outside the context of the program to the context of the underlying ADF [87]. In addition, Android is an event-driven system, which makes symbolic execution highly dependent on sequence of events; the symbolic execution engine has to wait for the user to interact with the system and tap on a button or initiate some other type of event for the program to continue the execution of a certain path. Furthermore, although Android apps are developed in Java, they run on Dalvik Virtual Machine (DVM) [22], instead of the traditional Java Virtual Machine (JVM). This is problematic, as current symbolic execution engines that are targeted at Java cannot be used for Android apps.

This chapter, presents SIG-Droid, an automated **S**ystem **I**nput **G**eneration framework for Android apps that tackles these challenges [83]. SIG-Droid combines program analysis techniques with symbolic execution [13] to systematically generate inputs for Android apps that achieve high code coverage. SIG-Droid leverages both *ATM* and *IM* to capture the behavior of an app.

SIG-Droid uses the *ATM* and the *IM* to exhaustively pinpoint possible ways an app can receive inputs. It then exchanges all concrete inputs with symbolic values, and gathers the constraints around those inputs. To determine the execution paths that should be symbolically analyzed, it automatically generates sequences of event handler methods from the inferred *ATM*. These sequences are called *Drivers*. Furthermore, to enable our symbolic execution engine to run the apps on JVM, and to resolve any possible external method calls resulting in path-divergence, models of Android library classes are created. After symbolically executing the app using the drivers, the solved values are used along with the corresponding events to create test inputs. The experiments corroborate SIG-Droid's ability to systematically generate test cases for end-to-end testing of Android apps that achieve high code coverage.

To summarize, the main contributions of the work presented in this chapter are:

- *Symbolic Execution for Android framework:* I provide solutions to solve the challenges of symbolic analysis for Android framework, and extend Symbolic Pathfinder (SPF) [65] to support Android apps (§ 5.2).
- *Implementation:* I develop a working implementation of SIG-Droid that automatically generates executable Robotium[31] test cases that include, among other things, event sequences and data inputs for testing of Android apps (§ 6.4).
- *Experiments:* I conduct experimental evaluation of our approach on both a collection of real-world Android apps and a benchmark-suite designed to assess the impact of input constraints and app's complexity on the effectiveness of system-level testing techniques. The results corroborate SIG-Droid's ability in achieving significantly higher code coverage compared to existing automated testing tools for Android (§ 6.5).

5.1 Overview of SIG-Droid

Figure 5.1 depicts a high level overview of SIG-Droid, which is comprised of three major components. The first component is the Model Generator that takes an app's source code and outputs two models, the *ATM* and the *IM*, as explained in details in Chapter 4.

The second component of SIG-Droid is the Symbolic Execution Engine. SIG-Droid is built on top of JPF, which uses the byte-code interpretation of the program under test.

Hence, the app's source code has to be compiled with Java compiler, instead of Android's Software Development Kit. This task is achieved by replacing platform-specific parts of the Android libraries that are needed for each app with stubs. These stubs are created in a way that each component's composition and callback behavior is preserved. This allows SIG-Droid to execute an Android app on JPF virtual machine without modifying the app's implementation.

The symbolic execution engine heavily utilizes the two generated models. The *ATM* is used to generate the app Drivers (i.e., use cases), while the *IM* is used to mark the input values that have to be exchanged with symbolic values. Furthermore, prior to running the symbolic analysis, the code is instrumented in order to track the sequence of events that occur in each path. The results are stored in the symbolic execution report that is used later in generating test cases.

Finally, the third component of SIG-Droid is the Test Case Generator. It takes the *IM* along with the symbolic execution report as inputs and generates test cases that can be executed on top of Robotium [31], which is an Android test bed. The focus of this Chapter is on generating test cases that achieve high code coverage, not on whether the test cases have passed/failed. Currently, collect two types of results are collected from the execution of tests: any exceptions that may indicate certain software faults as well as code coverage information. EMMA [88], an open source toolkit, is used for obtaining the statement coverage information. The next two sections describe the components of SIG-Droid in more detail.

5.2 Symbolic Execution for Android

Since Android apps are (1) event-driven, (2) prone to path-divergence, and (3) compiled into Dalvik byte-code, to build a symbolic execution engine for Android three major challenges have to addressed. This section explains how SIG-Droid's symbolic execution engine addresses these challenges.

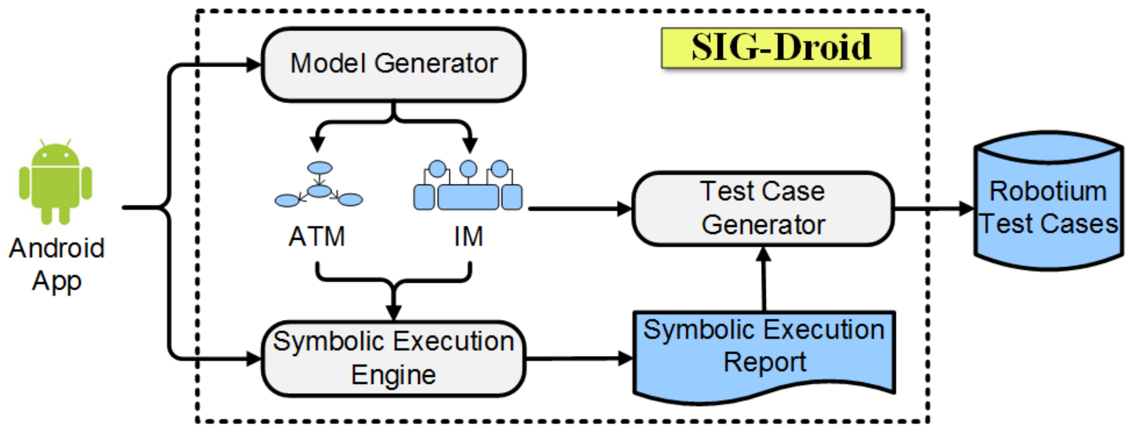

Figure 5.1: High level overview of SIG-Droid.

5.2.1 Handling Event-Driven Challenge

As Android is an event driven system, symbolic execution is highly dependent on events and their sequencing; meaning that the symbolic execution engine has to wait for the user to interact with the system and tap on a button or initiate some other type of event for the program to continue the execution of a certain path. Furthermore, the system itself or another application can initiate an event and cause the app to behave in a certain way.

To address this issue, symbolic execution engines, such as SPF [65], provide a mechanism to specify a Driver, which in the case of SPF is a Java program with a main method that contains the sequence of methods (event handlers) that should be used in a single run of the engine for determining the parts of the code that should be analyzed for gathering constraints. To generate the Drivers for Android apps, SIG-Droid uses the *ATM* as a finite state machine and traverse all the unique paths that do not contain a loop using a depth first search algorithm. This results in generating many possible sequences of events that represent possible use cases for the app.

As an example, using the *ATM* in Figure 4.1, if we start at `NewReportActivity.onCrea-te()` and follow through with `QuickReportAtivity.onCreate()` and `Next.onClick()`, we arrive at a plausible sequence. Clearly, if the app is comprised of more than one Activity and many events, the generated Driver would be more complex. Listing 5.1 illustrates a sample Driver for ERS generated in this way using the sample *ATM* of Figure 4.1. It contains two sequence of events, i.e., creates a `QuickReportAtivity` object by calling its constructor following by calling the `onCreate` method that triggers the start of the activity. Consequently, it simulates the action of user tapping on the *"Quick Report"* button by calling `onClick` method.

Note that since the *ATM* does not model the program's constraints, not all generated Drivers are necessarily valid sequences of events (i.e., can actually occur when the program executes). As will be detailed in Section 6.4, SIG-Droid does not use the Drivers for the purpose of generating the test cases, but only for the purpose of guiding the symbolic execution and solving the constraints on input values.

5.2.2 Handling Path-Divergence and Davlik Byte-Code Challenges

The second challenge is an Android program's dependence on framework libraries that make symbolic execution prone to path-divergence, and more so than traditional Java programs. In general, path-divergence occurs when a symbolic value flows outside the context of the program that is being symbolically executed and into the bounding framework or any external library [87]. Path-divergence leads to two major problems. First, the symbolic execution engine may not be able to execute the external library, as a result extra effort may be needed to support those libraries. Second, the external path may contain its own constraints that result in generating extra test inputs attempting to execute the diverged path rather than the program itself. This creates a scalability problem, as it entails symbolically executing parts of the Android operating system every time there is a path-divergence.

Indeed, in Android, path-divergence is the norm, rather than the exception. A typical Android app is composed of multiple Activities and Services communicating extensively with one another using Intents. An Intent is used to carry a value to another Activity/Service and as a result that value leaves the boundaries of the app and is passed through Android libraries before it is retrieved in the new Activity/Service.

Furthermore, Android apps depend on a proprietary set of libraries that are not available outside the device or emulator. Android code runs on Dalvik Virtual Machine (DVM) [22] instead of the traditional Java Virtual Machine (JVM). Thus, Android apps are compiled into Dalvik byte-code rather than Java byte-code. To symbolically execute an Android app using SPF, the app has to be transformed into the corresponding Java byte-code representation first.

To tackle the path-divergence problem and compile Android apps to Java byte-code, SIG-Droid provides its own custom built stub and mock classes. The stub classes are used to compile Android apps into JVM byte-code, while mock classes are used to deal with the path-divergence problem. I developed stubs that return random values within a reasonable range, when the return type of a method is primitive, and return empty instances of the object, when the return type is a complex data type. Dealing with Android platform, not only do I need to provide stub classes to resolve the byte-code incompatibility with JVM, but I also need to address the lack of Android logic outside the phone environment. Android uses its library classes as nuts and bolts that connect the different pieces of an app together.

A common instance of path-divergence in Android occurs when one Activity is initiated from another one and a value is passed from the source to the destination Activity. This process is performed by utilizing an Intent message (recall from Section 2.1 that in Android inter-component messaging is achieved through Intents). In the case of ERS, as shown in Figure 3.1, `NewReportActivity` uses the `startActivity` method of the Android library class `Activity.java` to start the app's `QuickReportActivity`. It creates an Intent in which the source and destination activities along with the values to be carried are specified. In this case, I provide the appropriate logic for Activity.java mock, such that when its `startActivity` method is called, the control flow moves to the `onCreate` method of the recipient activity.

Moreover, I create a mock for the `Intent.java` to address the path-divergence problem in cases where the payload is a symbolic value. As shown in Listing 3.2, an instance of Intent is passed to `startActivity`. If this Intent encapsulates a symbolic value for variable `amountValue`, it would result in path-divergence. To deal with this issue, I provided my

```java
public static void main(String[] args) {
   try {
      View v = new View(null);
      NewReportActivity newReport = new NewReportActivity();
      QuickReport i = newReport.new QuickReport();
      newReport.onCreate();
      i.onClick(v);
   }
   catch (Exception e) {
      e.printStackTrace();
   }
}
```

Listing 5.1: Sample Driver for ERS app.

own implementation of **putExtra** and **getExtra** methods in the mock implementation of **Intent.java**, such that the symbolic value of those variables is preserved. Android uses a **hashMap**⟨**String, Object**⟩ to store and retrieve the payload of an Intent, making it difficult to reason about a value stored as Object symbolically. To solve this problem, I provide my own implementation of a hash map that holds primitive values. Consequently, in our implementation of the **putExtra** and **getExtra** methods, I use our hash map implementation to enable the symbolic execution engine to reason about values that are exchanged using the Intent messages.

The last step prior to running symbolic execution of each app is to identify which values need to be executed symbolically. These are the values that the user can input using the GUI, e.g., the amount in the ERS app. As an example, for each input box in the *IM*, the source code of the corresponding activity is explored and the value of that input box, retrieved by calling **inputBox.getText()**, is exchanged with a symbolic value. It is important to keep a mapping between each introduced symbolic value and its corresponding widget on the screen. At the same time, the code is instrumented to record the sequence of actions taken. The mapping along with the sequence of actions captured in the *Drivers* are used by the test case generator to reproduce the values and actions in each test case.

```
<?xml version="1.0" encoding="utf-8"?>
<Report>
  <Path id="1"
    <Activity name=NewReportActivity>
      <MethodCall name=QuickReport.onClick()">
        <SoldvedVariable name="amountValue" value="1" />
      </MethodCall>
    </Activity>
  </Path>
  <Path id="2"
    <Activity name="NewReportActivity"
      <MethodCall name="QuickReport.onClick()">
        <SoldvedVariable name="amountValue" value="501" />
      </MethodCall>
    </Activity>
  </Path>
</Report>
```

Listing 5.2: Symbolic Execution report for Driver in Listing 1.

5.3 Test Case Generation

Following the extraction of models and symbolic execution of an app, SIG-Droid automatically generates test inputs that can be executed on an actual phone or emulator device. Running symbolic execution with each Driver results in a symbolic execution report. Each report specifies the concrete values that are obtained by solving the gathered symbolic conditions.

Each Driver representing a single path in the *ATM* may contain several constraints, thus it may result in multiple execution paths. For instance, if `amountValue` is a symbolic value in `NewReportActivity` in Figure 3.1, the Driver in Listing 5.1 would result in two different execution paths: One where the `amountValue` is less than \$500, and another where it is greater. Hence, the report for each Driver may result in several tests.

Moreover, as mentioned in Section 4.2, the *ATM* for each app only contains information about the possible chains of method calls regardless of constraints. As a result, the Drivers that are generated using the *ATM* may be invalid sequences of events, meaning that the

```
public class NewReportActivityTest_1 extends
  ActivityInstrumentationTestCase2<NewReportActivity> {

    private Solo solo;
       ...
       @smoke
       public void testMethod() throws Exception {
          solo.enterText(0,"501");
          solo.clickOnButton("Quick Report");
    }
    ...
}
```

Listing 5.3: Code snippet of a Robotium test automatically generated by SIG-Droid for NewReportActivity.

constraints may prevent the execution of certain events. In order to make sure that I only generate valid sequences of events in each test case, the code is instrumented to track the actual method execution sequence during the symbolic execution. Thus, the symbolic execution report contains the sequence of called methods as well. Listing 5.2 shows the symbolic execution report for the Driver of Listing 5.1.

Since the report contains only the event handlers and not the actual event generators (e.g., the *ID* of the buttons on a screen), to generate test cases SIG-Droid uses the *IM* to determine the event generator corresponding to each event handler in the report. For example, `QuickReport.onClick` handler method in Listing 5.2 is the handler for the *"Quick Report"* button on `NewReportActivity` screen of Figure 3.1.

Listing 5.3 illustrates one of the Robotium test cases generated by SIG-Droid that corresponds to the report shown in Listing 5.2. Solo is a Java class provided by Robotium that executes the test (essentially represents the user of the app). This test case inputs `5001` in the `amount` text box, which has the index of zero, meaning it is the first text box on that activity, and then clicks on *"Quick Report"* button.

5.4 Evaluation

To evaluate SIG-Droid, three research questions are formulated:

- **RQ1:** Is SIG-Droid capable of generating test cases for real-world Android apps?
- **RQ2:** How well does SIG-Droid perform? Can SIG-Droid achieve a better code coverage than state-of-the-art Android system testing frameworks?
- **RQ3:** How scalable is the approach in generating test cases for complex applications, i.e., apps involving highly constrained input values?

For investigating RQ1, I apply SIG-Droid to several real-world apps from an open-source repository, called F-Droid [89]. These apps are picked based on the following criteria: (1) the source code for the applications must be available (2) the app only uses standard GUI widgets that are included in Android API and does not use any third party widgets, and (3) the apps should capture the different application categories, such as productivity, entertainment, and tools. Moreover, SPF does not handle anonymous classes. As a result, I refactored the source code of the apps to ensure they do not contain any anonymous classes.

Table 5.1 lists these apps. LOC, Activities, and Category columns report lines of code, number of activities of each app, and category of each app, respectively. [1]

For addressing RQ2, I compare SIG-Droid with two approaches: Android Monkey [37] and Dynodroid [8]. I also considered other testing tools for the evaluation, but were not able to include them for various reasons. Some focus on other objectives (e.g., A^3E [41] focuses on discovering Activities by covering a model of an app and does not report statement coverage), while there were practical difficulties with others (e.g., SwiftHand [5] exits with an exception when used on our apps).

For answering RQ3, I develop a benchmark-suite that entails a collection of synthetic

[1]Per our study of 100 F-Droid apps, average number of activities for an app is 4.

Table 5.1: Open-source apps used in the evaluation.

App	LOC	Activities	Category
CalAdder	276	2	Productivity
Tipster	501	1	Tool
MunchLife	631	2	Entertainment
JustSit	849	4	Productivity
AnyCut	1095	4	Tool
TippyTipper	2953	6	Tool

apps in different levels of complexity. To measure apps' complexity, I use three well-established complexity metrics from literature, namely *Method Call Sequence Depth*, *McCabe Cyclomatic Complexity*, and *Block Depth per Method*. I then measure the impact of input constraint and complexity on scalability of our technique.

All experiments were conducted on an Apple iMac machine with 8GB memory and a dual core 2.4GHz processor. I used Android Virtual Devices (Android emulators) with 1GB RAM and 2GB SD Card. A fresh emulator was created for each app along with only default system applications. During the experiments, I used EMMA [88] to monitor the statement coverage. The reported line coverage is gathered by running all of the generated test cases on each app.

5.4.1 Experiment 1: Open-Source Apps

In our first set of experiments, I measured and compared the source code statement coverage achieved using the test cases generated by SIG-Droid, Monkey, and Dynodroid. Android Monkey, developed by Google, is essentially a fuzzing tool that sends random inputs and events to the app under test. Dynodroid improves on the number of inputs/events Monkey uses, thus achieves a similar coverage with less generated events. Since both Dynodroid and Monkey treat both data widget inputs and events as input events, to achieve a fair

Table 5.2: Comparison of SIG-Droid with other techniques

App	**Number of Events**	**SIG-Droid**		**Monkey**		**Dynodroid**		**Monkey (2000 Events)**		**Dynodroid (2000 Events)**	
		Coverage	Time(sec)	Coverage	Time(sec)	Coverage	Time(sec)	Coverage	Time(sec)	Coverage	Time(sec)
CalAdder	6	82%	122	10%	37	-	-	35%	118	-	-
Tipster	35	83%	159	31%	26	53%	462	67%	104	59%	33825
MunchLife	20	74%	186	36%	44	32%	354	49%	75	54%	31421
JustSit	30	75%	137	20%	46	26%	335	35%	163	53%	41252
AnyCut	18	79%	179	6%	58	38%	282	6%	71	66%	21757
TippyTipper	91	78%	484	26%	45	-	-	42%	106	-	-

comparison among Android Monkey, Dynodroid, and SIG-Droid, I ran each tool with the same number of events. To be more specific, I counted the number of events used in SIG-Droid generated test cases, and used that number as the number of inputs for Monkey and Dynodroid. As both Monkey and Dynodroid are based on random approaches, using the same low number of events that are generated by SIG-Droid may not be fair for a comparison. To address that, I also run both tools with 2,000 input events, which is the maximum number possible for Dynodroid [8].

The line coverage results are summarized in Table 6.2.[2] Column *# of Events* represents the number of input events in Robotium test cases generated by SIG-Droid. An event in a Robotium test case can be either an action, such as a button click, or entering an input value into a widget like a text box. The next six columns then represent the line coverage and the time taken to generate and execute test cases for SIG-Droid, Monkey, and Dynodroid, given the number of input events shown in the first column for each app. Columns *Monkey(2000 Events)* and *Dynodroid(2000 Events)* represent the same information, but when Monkey and Dynodroid are given 2,000 events.

In terms of the total code covered for each app, SIG-Droid easily outperforms both Dynodroid and Monkey, achieving higher coverage for all apps. Given the same number of input events shown in the first column for each app, SIG-Droid's coverage on average outperforms Monkey and Dynodroid by a 57% and a 41% margin, respectively. Even when the other tools are allowed to use more events, the code coverage achieved by them is still clearly outperformed by SIG-Droid's. In addition, SIG-Droid runs 2X faster than Dynodroid, and about 3X slower than Monkey. This is not surprising, given that Monkey is a completely random testing tool. More specifically, when Monkey traverses a path, it does not backtrack or use any other systematic way to test the app. Therefore, Monkey's test coverage does not considerably improve even with significantly higher number of input events. In contrast, SIG-Droid relies on the *ATM* to generate input events, thereby leads to unique sequences of events that cover nodes captured in the *ATM*.

[2] I could not run Dynodroid on CalAdder and TippyTipper as Dynodroid runs on Android 2.3 and it does not support apps developed with newer APIs.

Although SIG-Droid performs significantly better than the two mentioned methods, it fails to achieve complete code coverage. In some cases this could be due to unreachable code or pieces of code that handle specific events while the app is running, such as phone unlock. It is also partially attributed to the fact that the current implementation does not handle loops in the *ATM* as well as well-known symbolic execution shortcomings in dealing with non-primitive data-types. Additionally, our program analysis does not support all possible ways that Android apps could be developed. For instance, there are many ways of handing events in Android, and one of those is through inline class declarations, which SIG-Droid does not support. The results show that SIG-Droid is significantly more effective than existing system testing techniques targeted at Android apps.

5.4.2 Experiment 2: Benchmark Apps

In practice, symbolic execution is predominantly known to be suffering from scalability issues caused by problems such path-explosion [90]. To assess SIG-Droid's performance and scalability, I needed a way of selecting benchmark apps that are nontrivial. Finding real-world apps that fall into a variety of categories defined by a number of complexity metrics is nontrivial. As shown previously [91,92], an effective way to address this problem is to write benchmark applications that satisfy the requirements. Similarly, I built an Android app generator that produces apps with different levels of complexity for our experiments. These apps provide us with a controlled environment, i.e., these apps do not contain features, such as loops in their control flow, that are not fully supported by symbolic execution tools, including SPF. By using these apps for benchmarking the performance and scalability of SIG-Droid, I can remove the impact of such known limitations and only concentrate on the impact of app complexity. However, I also needed a way of ensuring the synthesized apps were representative of real apps.

To that end, I first conducted an empirical study involving real world apps and analyzed approximately 100 apps chosen randomly from F-Droid repository [89]. The selected apps were in various categories, such as education, communication, gaming, etc. I analyzed these

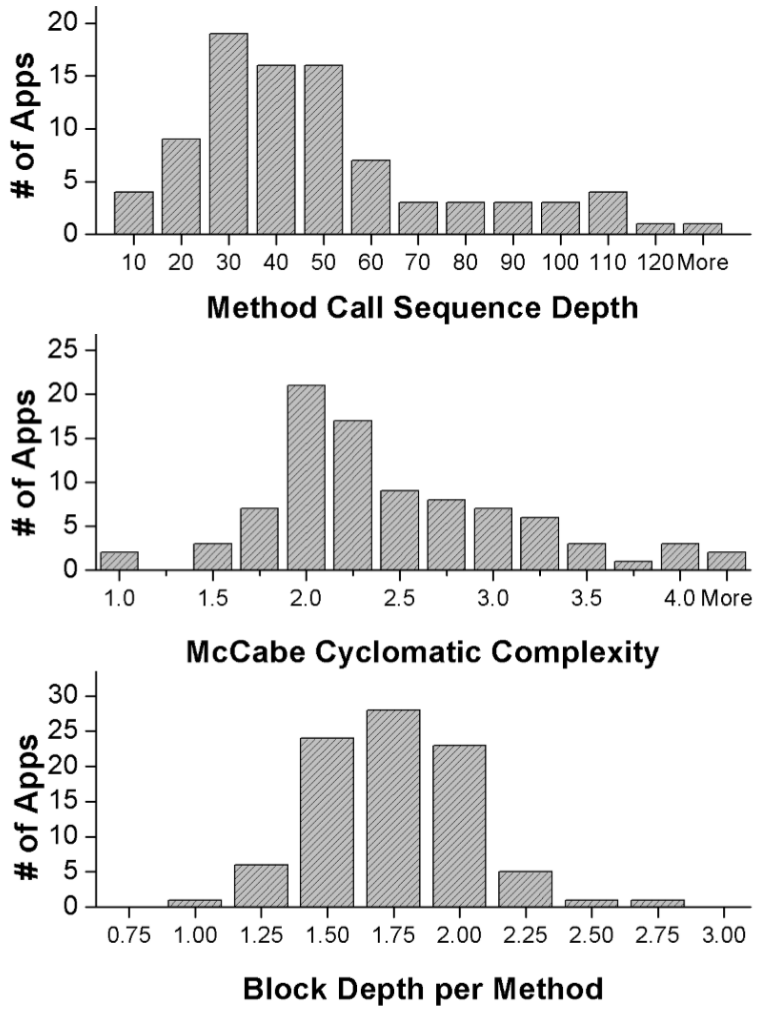

Figure 5.2: Android complexity metrics distribution from a random sample of 100 apps.

apps according to three major complexity dimensions that could impact SIG-Droid: (1) Method Call Sequence Depth — the longest method call sequence in the app, (2) McCabe Cyclomatic Complexity — the average number of control flow branches per method, and (3) Block Depth per Method — the average number of nested condition statements per method. Figure 5.2 shows the distribution of these complexity dimensions among the 100 Android apps from F-Droid. Our app generator is able to synthesize apps with varying values in these three dimensions.

I then defined *complexity classes* for generating subject apps in our experiments. For

Table 5.3: Benchmark apps.

App	Max Method Call Sequence	McCabe Cyclomatic Complexity/Method	Nested Block Depth/Method	Symbolic Exec. Time(sec)	Overall Exec. Time(sec)
1	15.00	1.6590	1.2929	2	40
2	23.60	1.8506	1.4018	2	37
3	27.10	1.9332	1.4742	4	175
4	32.00	2.0416	1.5216	4	135
5	38.00	2.1945	1.5650	6	30
6	41.40	2.3606	1.6860	5	60
7	50.10	2.5575	1.7761	5	299
8	62.20	2.8956	1.8850	6	596
9	90.80	3.2287	1.9867	6	446

that, I aggregated the data collected through our empirical study, as shown in Figure 5.2, and produced the overall app complexity classes ranging from 10th to 90th percentile, shown in Table 5.3. For instance, the 10th overall complexity in Table 5.3 corresponds to the 10th percentile in all of the three dimensions shown in Figure 5.2. Essentially this means that an app belonging to a lower class is less complex with respect to all three dimensions compared to an app from a higher class.

I used SIG-Droid to test one generated app from each of the nine complexity classes. I evaluated SIG-Droid by measuring the execution time and the resulting statement coverage. By conducting this experiment, I were able to measure the impact of input constraint and complexity on performance of our technique.

Table 5.3 shows the symbolic execution time as well as the overall execution time, which includes the symbolic execution time and the time it took to generate and execute the tests. As one would expect, the increase in the app complexity results in a modest increase in the symbolic execution time, since more constraints need to be solved. I also notice an increase in the overall execution time, due to the higher number of generated test cases and consequently the time needed for their execution. The results demonstrate that SIG-Droid is capable of scaling to even the most complex Android apps. Although not the focus of this experiment, it corroborates our earlier assertion that SIG-Droid's inability to obtain

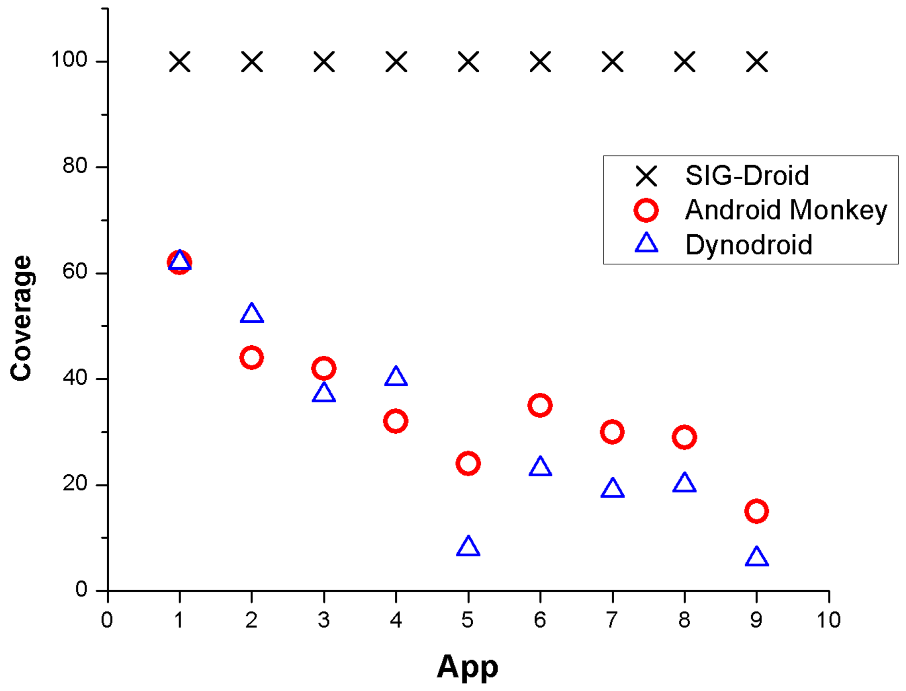

Figure 5.3: Coverage for Benchmark Apps.

complete code coverage in the case of real apps is due to the existence of unreachable code as well as incomplete model of app behavior. More advanced program analysis techniques for obtaining complete model of app behavior could reduce the gap between SIG-Droid's actual code coverage in real apps and its theoretical potential in synthesized apps.

Finally, SIG-Droid is benchmarked against Monkey and Dynodroid using the synthetic Android apps. Figure 5.3 shows the coverage results for this experiment. As the complexity of the apps increases, the coverage for both Android Monkey and Dynodroid drops significantly. However, SIG-Droid's code coverage does not suffer from the increase in app complexity. This is mainly because it uses the *ATM* to generate unique sequences of events along with symbolic execution to solve the input constraints. Accordingly, the increase in the app complexity has no negative impact on the performance of SIG-Droid.

5.5 Conclusion

This chapter presented SIG-Droid, a novel framework for automated testing of Android apps. The key contributions of this work are (1) a symbolic execution engine that supports Android apps, (2) combining model-based testing with symbolic execution to systematically generate test inputs for Android apps, and (3) a supporting framework that generates effective system level test inputs for Android apps.

Although SIG-Droid has shown to be significantly better than random tools for automated testing it is not free of shortcomings. First, expanding support for Android libraries through the development of additional stubs and mock classes requires significant manual engineering effort. Moreover, SIG-Droid's program analysis does not support inline declaration of event handlers to generate more accurate model of app behavior. Finally, SIG-Droid currently only focuses on generating values for basic GUI data-input widgets. Hence, there is a need for a more comprehensive technique that can be applied to a more diverse set of apps. The next chapter aims to address these limitations by employing another well-known input generation technique for Android apps.

Chapter 6: Input Generation for Android Applications Using Combinatorial Testing

Testing of GUI-driven apps requires utilizing a large number of event sequences. These sequences are often generated by GUI interactions involving radio-boxes, check-boxes, drop-down lists, etc. Exhaustive combinatorial testing [20], tries all possible GUI combinations which is often computationally prohibitive. Although t-way testing produces a smaller number of tests, it is also less effective than exhaustive testing in terms of both code coverage and fault detection. For instance, when pairwise testing is used, the parts of code that depend on the interaction of three or more GUI widgets may remain uncovered.

To illustrate the challenges of combinatorial testing, consider a situation in which the user clicks on the *ItemizedReport* button of *NewReportActivity* and subsequently on the *Next* button of *ItemizedReportActivity* (see Figure 3.1). *NewReportActivity* contains the *Destination* drop-down list with 10 choices, and the *Currency* check-box with 3 exclusive choices. Let us also assume two values of 100 and 0 have been identified as proper input classes for the *Amount* field. This would result in a total of $10 \times 3 \times 2 = 60$ unique combinations for *NewReportActivity.* Similarly, *ItemizedReportActivity* contains the *Total Days* drop-down list with 6 choices, the *First Day Meals* and *Last Day Meals*, each of which has 3 inclusive choices, resulting in a total of $6 \times 2^3 \times 2^3 = 384$ unique combinations.

Since the widget values selected on one Activity could impact the behavior that is manifested in subsequent Activities, for GUI system testing, I also need to consider the interaction of widgets across Activities. Thus, the number of all unique tests for the above use case is $60 \times 384 = 23,040$. The number of tests would continue to grow if I consider the other Activities comprising this app. This approach is infeasible in practice, in terms of both the effort required to execute the tests and the effort required in assessing the results.

The main goal of the research presented in this chapter is to drastically reduce the number of tests for achieving a comparable coverage as exhaustive GUI testing. The insight guiding this research is that not all GUI widgets and actions interact with one another.

To that end, the control- and data-flow dependencies among the GUI widgets, event handlers, and Activities of an app, can be statically extracted. This can be done without access to the source code, and rather from the app's APK file.

An example of GUI widget interaction can be gleaned from Figure 3.1a. Here, I can see that when *Total Days* obtains a value of 1, *Last Day Meals* check-boxes are disabled, thus indicating a dependency between these two widgets, implying that their combinations should be tested. On the other hand, if the analysis indicates that *Total Days* and *First Day Meals* are indeed independent of one another, I can safely conclude that their combinations do not need to be tested. Such dependencies, provide the basis for combinatorial generation of tests.

To appreciate the significant reductions possible this way, consider the use case of ERS described earlier. By identifying the dependencies between the input-widgets in the code, I can only generate $max\{(6 \times 2^3), 2^3\} = 48$ tests for the *ItemizedReportActivity* when the *Next* button is clicked. That represents a reduction of 336 combinations compared to the exhaustive approach. The dependencies imply that the $(6 \times 2^3) = 48$ possible combinations for *Total Days* and *Last Day Meals* are independent of the $2^3 = 8$ possible combinations for *First Day Meals*. Since I can use combination of independent widgets in the same test, the dependent widgets with the biggest number of unique combinations determine the number of generated tests. Here, the 48 combinations for *Total Days* and *Last Day Meals* are merged with the 8 combinations for *First Day Meals* to produce 48 widget combinations for the *ItemizedReportActivity*. For testing both activities together, only $60 \times 48 = 2,880$ tests are produced, representing a reduction of $20,160$ tests compared to the exhaustive approach.

Assuming the static analysis has produced an accurate representation of dependencies, the reduced set of tests generated using this technique would be as effective as exhaustively generated tests in terms of their coverage and fault detection power.

This Chapter presents TrimDroid (**T**esting **R**educed GU**I** Co**M**binations for And**DROID**), a fully-automated combinatorial testing approach for Android apps [93]. Given an Android APK file, TrimDroid employs static analysis techniques that are informed by the rules and constraints imposed by the Android ADF to identify GUI widgets that interact with one another.[1] Thus, the set of interacting widgets become candidates for t-way combinatorial testing. By avoiding the generation of tests for widgets that do not interact, TrimDroid is able to significantly reduce the number of tests. For identifying the interactions, TrimDroid statically analyzes the control- and data-flow dependencies among the widgets and actions available on an app. Finally, TrimDroid uses an efficient constraint solver to enumerate the test cases covering all possible combinations of GUI widgets and actions.

The evaluation results for TrimDroid show that it achieves the same coverage as exhaustive combinatorial testing, but reduces the number of test cases by 57.86% on average and by as much as 99.9%. This reduction is important, as it not only reduces the time it takes to execute the test cases, but also significantly decreases the effort required to inspect the test results, which is often performed by a human engineer.

6.1 Approach Overview

Figure 6.1 depicts a high-level overview of TrimDroid, which is comprised of four major components: *Model Extraction*, *Dependency Extraction*, *Sequence Generation*, and *Test-Case Generation*. Together, these components produce a significantly smaller number of test cases than an exhaustive combinatorial technique, yet achieve a comparable coverage.

Similar to our previous work [47], *Model Extraction* produces two types of models by statically analyzing an Android app:

- Interface Model (*IM*) provides a representation of all the GUI inputs of an app, including the input widgets and events (actions) for each Activity. TrimDroid uses the IM to determine how a GUI screen can be exercised in order to generate the tests

[1]An APK file is a Java bytecode package used to install Android apps.

for it.

- Activity Transition Model (*ATM*) is a finite state machine representing the event-driven behavior of an Android app, including the relationships among its Activities and their event handlers (transitions). Since my research targets GUI testing, I only extract information that is related to Activities, not other Android components (e.g., Services). Figure 4.1 depicts the *ATM* for the entire ERS app.

These models are represented in Alloy [21], a formal specification language based on first order relational logic. Alloy specifications can be analyzed using Alloy Analyzer, thereby allowing us to systematically explore the combinatorial space with the help of a constraint solver.

In a step parallel to *Model Extraction*, *Dependency Extraction* identifies GUI-induced dependencies among app elements using a combination of control- and data-flow analysis techniques. *Dependency Extraction* identifies three types of dependencies (1) when one GUI widget depends on the value of another widget, e.g., a drop-down menu is disabled, because a check-box is not selected, (2) when a GUI event handler depends on a widget value, e.g., a button handler method uses the selected value of a check-box, and (3) when an Activity depends on the widget values from a preceding Activity, e.g., the widget values from a preceding Activity are included in the payload of an *Intent* starting a new Activity.[2] These dependencies are also represented in the form of Alloy specifications and used by *Test-Case Generation* in a later step for pruning the combinatorial space of tests.

Sequence Generation uses the Alloy Analyzer to synthesize sequences of events that cover the paths in the *ATM*. Each path in the *ATM* represents a sequence of events in a possible use case. A good coverage of the *ATM* is essential for achieving high code coverage. TrimDroid covers the paths using the *prime path* coverage criterion, known to subsume most other graph coverage criteria [56]. The coverage of a technique t_1 subsumes the coverage of a technique t_2 if and only if 100% coverage for t_1 implies 100% coverage for t_2 [94].

[2]All Android components are activated via *Intent* messages. An Intent message is an event for an action to be performed along with the data that supports that action.

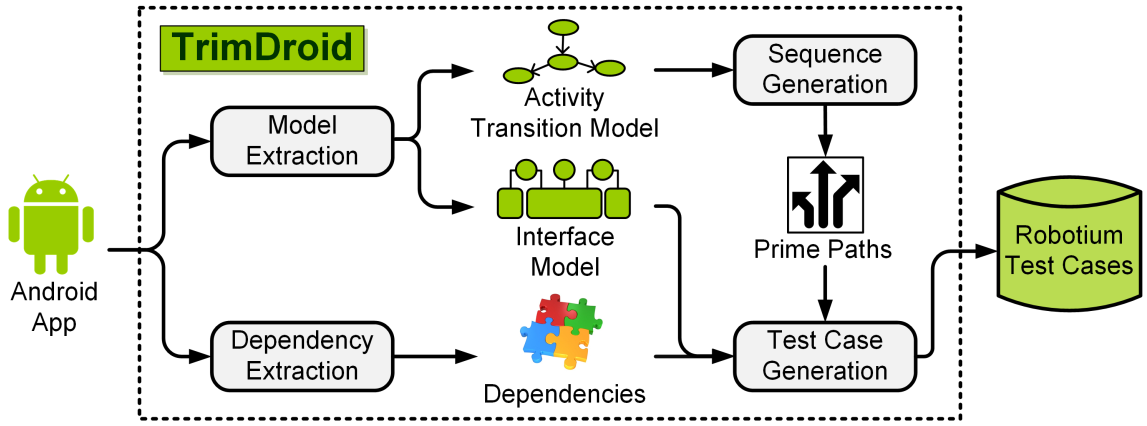

Figure 6.1: A high-level overview of TrimDroid

Finally, *Test-Case Generation* constructs system tests by performing three key steps. First, it traverses the sequences of events representing the paths produced by *Sequence Generation*. Second, for each step in a given sequence, it uses Alloy Analyzer to generate value combinations for different GUI widgets. To that end, *Test-Case Generation* utilizes (1) the sets of dependent widgets generated by *Dependency Extraction* and (2) the specification of each widget in the *IM*. Lastly, *Test-Case Generation* merges the value combinations to create tests that cover the entire sequence of events in each path of the ATM. The generated tests can then be executed using Robotium [31], an Android test-automation framework.

The next four sections describe the four components of TrimDroid in more detail.

6.2 Dependency Extraction

TrimDroid uses the dependencies among the app elements to determine the combinations that should be tested, and those that can be safely pruned. To that end, *Dependency Extraction* determines three types of dependencies as described further below.

6.2.1 Widget Dependency

Two widgets $w1$ and $w2$ are dependent if combinations of their values affect an app's control- or data-flow. Widget combinations that affect the control-flow impact the code coverage of generated tests; widget combinations that affect the data-flow determine the state of the system under test. Here are two possible dependencies between $w1$ and $w2$ that TrimDroid detects:

(Case 1) $w2$'s use depends on the value of $w1$. This can occur in two situations. First, a widget $w1$ is used in a conditional statement, and widget $w2$ is used along either branch of that statement. An example of the first case is shown below, where `lastBreakFast` is dependent on `totalDays`:

```
if((String.valueOf(totalDays.getSelectedItem())).equals("1")) {
    lastBreakFast.setEnabled(false);
}
```

The second situation occurs when the value of widget $w1$ affects a reference r, and $w2$'s use depends on the value of r. An example of this case is shown below, where the use of `totalDays` is indirectly dependent on `firstBreakfast` based on the variable `mealsCount`:

```
  if(firstBreakFast.isChecked())
    mealsCount++;
  if(mealsCount > 0)
    totalMeals = totalDays.getValue()*3 + mealsCount;
```

(Case 2) In a conditional statement, widget $w1$ is used and reference r is defined in its block, and r is later used in the block of another conditional statement, where $w2$ is used.

Algorithm 2: *wDep*

```
Input: $a \in A$
Output: $WD \subset \mathcal{P}(widgets(a))$
1  $WD \leftarrow \emptyset$;
2  $depPairs \leftarrow \emptyset$;
3  foreach $meth \in a.Methods$ do
4      foreach $w$ used in a conditional statement $stmt_1$ of $meth$ do
5          if $isAWidget(w)$ then
6              foreach $w_{1a}$ used along either branch of $stmt_1$ do
7                  if $isAWidget(w_{1a})$ then
8                      $depPairs \leftarrow depPairs \cup \{\{w_{1a}, w\}\}$;
9              foreach $r_v$ defined along either branch of $stmt_1$ do
10                 foreach $w_2 \in widgetsWhoseValueAffects(r_v)$ do
11                     $depPairs \leftarrow depPairs \cup \{\{w_2, w\}\}$;
12                 foreach conditional statement $stmt_2$ that uses $r_v$ do
13                     foreach $w_{1b}$ used along either branch of $stmt_2$ do
14                         if $isAWidget(w_{1b})$ then
15                             $depPairs \leftarrow depPairs \cup \{\{w_{1b}, w\}\}$;
16 $WD \leftarrow merge(depPairs)$;
17 $WD \leftarrow WD \cup isolateRemainingWidgets(WD, widgets(a))$;
```

An example of this case is shown below, where the value combinations of `firstBreakfast` and `firstLunch` impact the value of `mealsCount`:

```
if(firstBreakFast.isChecked())
  mealsCount++;
if(firstLunch.isChecked())
  mealsCount++;
```

Algorithm 2 defines *wDep*, which partitions $widgets(a)$ based on the two cases above. The algorithm takes an Activity a as input and produces WD, a partition for $widgets(a)$ where $WD \subset \mathcal{P}(widgets(a))$.

For each method, *wDep* iterates over each reference w that is used in a conditional statement and determines if w refers to a widget (lines 3–5 of Algorithm 2). To make that determination, $isAWidget(w)$ traverses the definition-use chain of w to determine if any of its definitions refers to a widget. At this point, *wDep* distinguishes the two cases that result in widget dependencies.

To determine the first situation of Case 1, *wDep* checks if any other variable w_{1a} is used and references a widget (lines 6–7 of Algorithm 2). If so, *wDep* creates a widget dependency $\{w_{1a}, w\}$ (line 8 of Algorithm 2).

To obtain widget dependencies for Case 2, *wDep* identifies any variable r_v defined after the conditional statement where w is referenced (line 9 of Algorithm 2). For any r_v whose value is affected by widget w_2 (line 10 of Algorithm 2), *wDep* creates the widget dependency $\{w_2, w\}$ (lines 11 of Algorithm 2). Here, $widgetsWhoseValueAffects(r_v)$ returns the widgets used in a conditional statement whose value affects a reference r_v by traversing r_v's definition-use chain.

To identify the second situation of Case 1, *wDep* further checks if reference r_v is used in a second conditional statement (line 12 of Algorithm 2). If a reference to a widget w_{1b} is used along either branch, then *wDep* creates a widget dependency $\{w_{1b}, w\}$ (lines 13–15 of Algorithm 2).

The widget dependency pairs are then merged (line 16). Two widget dependency pairs are merged if any of their elements intersect. For instance, two dependency pairs $\{w_\alpha, w_\beta\}$ and $\{w_\beta, w_\omega\}$ are merged, and the resulting set $\{w_\alpha, w_\beta, w_\omega\}$ is stored in *WD*. Finally, widgets that do not interact with any other widget, and thus not part of any dependency pair set, are each isolated into their own singleton set and added to *WD* (line 17).

6.2.2 Handler Dependency

To further reduce the number of test cases, *Dependency Extraction* identifies the dependencies between widgets and event handlers. This kind of dependency occurs when a widget value is used in an event handler, indicating that all combinations of the widget and the

Algorithm 3: *hDep*

```
Input: $a \in A, e \in eHandlers(a)$
Output: $HD \subset \mathcal{P}(widgets(a))$
1 $HD \leftarrow \emptyset$;
2 foreach $r$ is used in $e$ do
3     if $isAWidget(r)$ then
4         foreach $wd \in wDep(a)$ do
5             if $r \in wd$ then
6                 $HD \leftarrow HD \cup \{wd\}$;
7                 break;
```

event resulting in the invocation of event handler should be tried. As an example of this, consider the following code snippet, where the value of `totalDays` is used in the `onClick()` method of `NextButton` in the *ItemizedReportActivity* of ERS:

```
public class NextButton implements OnClickListener {
  public void onClick(View v) {
    totalDaysValue = String.valueOf(totalDays.getSelectedItem());
    ...
  }
}
```

If an event handler uses multiple widgets, all combinations of those widgets according to their widget dependencies need to be tested together with the handler's event.

Thus, the pruning of irrelevant test combinations is achieved through determining the widgets that are not used by event handlers. For example, the `onClick()` handler for the *Reset* button of *ItemizedReportActivity* clears the screen regardless of the values of the widgets. Hence, no value combinations of the widgets on *ItemizedReportActivity* need to be tested with the *Reset* button.

Algorithm 3 defines *hDep*, which partitions *widgets*(a) into a set based on handler dependencies. The input to the algorithm is an activity a and an event handler e. The output of the algorithm is HD, a partition of $widgets(a)$ where $HD \subset \mathcal{P}\left(widgets(a)\right)$.

6.2.3 Activity Dependency

The third type of dependency involves the widget values in one Activity that may impact the behavior of another Activity. If so, we need to test all combinations of those widgets in a first Activity impacting a second activity with all combinations of widgets in the second Activity. Since Activities in Android communicate using Intent messages, I say the value of a widget w in an Activity a_i may impact another Activity a_j, if it affects the payload of an Intent that is sent from a_i to a_j. For example, as shown in the following code snippet, *NewReportActivity* sends an Intent that starts the *ItemizedReportActivity* and passes the selected value for `currencyRB` in the payload of the Intent:

```
public class ItemizedReportButton implements OnClickListener {
  public void onClick(View v) {
    int selectedId = currencyRB.getCheckedRadioButtonId();
    currencyRB = (RadioButton)findViewById(selectedId);
    String currency = currencyRB.getText();
    Intent intent = new Intent(this,ItemizedReportActivity.class);
    intent.putExtra("currency", currency);
    startActivity(intent);
  }
}
```

On the other hand, from the above code snippet, we can see that the value of *Destination* drop-down menu from *NewReportActivity* does not impact the Intent sent to *ItemizedReportActivity*. Thus, there is no need to test all combinations of widgets on *NewReportActivity*

Algorithm 4: *aDep*

```
Input: $a_i, a_j \in A, e \in eHandlers(a)$
Output: $boolean$
/* get the intent sent from $a_i$ to $a_j$ in $e$                     */
1 $I \leftarrow a_i.getIntent(e, a_j)$
2 foreach $payload \in I.IntentExtras()$ do
3     $refs \leftarrow getAffectedReferences(payload)$
4     foreach $r \in refs$ do
5         if $isAWidget(r)$ then
6             return true;
7 return false;
```

with the widgets on *ItemizedReportActivity*. This provides us with yet another opportunity to prune the tests.

Algorithm 4 defines *aDep*, which determines whether an Activity has a dependency to widget values selected in a preceding Activity. The algorithm takes two Activities a_i and a_j, corresponding to the source and destination of an Intent, an event handler e, realizing the transition between the two activities, and returns true if an Activity dependency exists and false otherwise.

6.3 Sequence Generation

In GUI system testing, a test is comprised of two parts: sequence of events (e.g., button clicks) and selection of input values (e.g., drop-down menu choices). This section describes how TrimDroid produces sequences of events that represent possible use cases for the system. The next section provides the details of how the dependencies are used to determine the combination of input values for each sequence of events.

TrimDroid's approach for the generation of event sequences is based on using a formal language to describe the *ATM* as well as the coverage criteria for traversing it. TrimDroid then uses an automated constraint solver to exhaustively synthesize the space of possible paths. Each path in the *ATM* represents a sequence of event handlers triggered in a possible use case for the system. These paths can be generated using any given coverage criteria (e.g., node coverage, edge-pair coverage). TrimDroid relies on *prime path coverage* as it has been shown to *subsume* most other graph coverage criteria [56]. A coverage criterion α subsumes coverage criterion β, if and only if 100% α coverage implies 100% β coverage [94].

TrimDroid represents an *ATM* in the form of an Alloy model [21]. Alloy is a formal modeling language with a comprehensible syntax that stems from notations ubiquitous in object orientation, and semantics based on the first-order relational logic [21], making it an appropriate language for declarative specification of both application models and properties to be verified. Listing 6.1 shows (part of) the Alloy specification of ATM, specifically the signatures for `activity`, `simplePath` and `primePath`. Each Activity has a set of event handlers (`eHandlers`), and a field (`isStart`), indicating whether it is a starting activity or not. Lines 4–11 present the `simplePath` signature along with its *facts* that specify the elements involved in, and the semantics of, a *simple path*, respectively. A simple path is a sequence of *transitions* from the starting activity (i.e., a_0), where no activity node appears more than once, except possibly when the first and last nodes are the same. A *prime path* then, as specified in lines 12–14, is a simple path that does not appear as a proper sub-path of any other simple path. An example of a prime path in the *ATM* of Figure 4.1 is: $a_0 \xrightarrow{e_1} a_2 \xrightarrow{e_4} a_3 \xrightarrow{e_5} a_4$.

A test path satisfies prime path coverage if and only if it starts from a starting node and ends in a final node while covering a prime path in a graph [56]. The prime-path criterion limits visits of each loop to one, since simple paths have no internal loops. It also limits the number of generated paths, as it only contains paths that are not sub-path of any other path. The *ATM* of the running example (see Figure 4.1) thus includes three prime paths, automatically generated using Alloy Analyzer.

```
abstract sig activity{
  isStart: one IsStart,                    // indicator of a starting activity
  eHandlers: set activity }                       // activity's event handlers
sig simplePath{
  first: one activity,                               // the starting activity
  transitions: activity−>activity }{              // sequence of transitions
    first.isStart = Yes
    first in transitions.activty
    all x: activity | lone x.transitions
    transitions in eHandlers
    no x: transitions.activity| x !in first.∗(transitions) }
sig primePath extends simplePath{ }{    // an activity with no outgoing transition
  no finalActivity[transitions].eHandlers }
fun finalActivity[r: activity−>activity] : one activity {
  r[activity]−r.activity }
```

Listing 6.1: Specifications for *ATM* in Alloy.

6.4 Test Generation

Each system test $st \in ST$ is comprised of a sequence of *activity tests*: $st = \langle at_{a_0}, at_{a_1}, ..., at_{a_n}\rangle$. An *activity test* consists of widget value combinations and an event that exercises a particular Activity and results in a transition, either to itself or to another Activity, according to *ATM*.

TrimDroid generates the system tests in two steps. First, it generates the activity tests using the widget value combinations and events available on each Activity. Afterwards, it combines activity tests into a sequence that represents a GUI system test to cover a particular prime path.

To illustrate this process, I use the execution scenario for the ERS app shown in Figure 3.1. The ATM for ERS (recall Figure 4.1) shows its five activities (denoted as a) and their transitions (denoted as e). The input classes for widgets on `NewReportActivity` are captured in Figure 6.2a, where $ic(w)$ indicates the possible values for widget w. Figure 6.2b captures the same information for `ItemizedReportActivity`. In addition, the widget dependencies (recall Algorithm 2), handler dependencies (recall Algorithm 3), and activity

NewReportActivity			
#	$ic(dest)$	$ic(amount)$	$ic(cur)$
1	Rome	100	Euro
2	London	0	Dollar
3	Rome		Pound
.	.		
.	.		
10	Berlin		

(a) input classes of widgets in `NewReportActivity`.

ItemizedReportActivity							
#	$ic(totalDays)$	$ic(fbf)$	$ic(fl)$	$ic(fd)$	$ic(lbf)$	$ic(ll)$	$ic(ld)$
1	1	true	true	true	true	true	true
2	2	false	false	false	false	false	false
3	3						
.	.						
.	.						
6	6						

(b) input classes of widgets in `ItemizedReportActivity`.

Dependencies
$wDep(a_0) = \{\{dest, cur\}, \{amount\}\}$
$wDep(a_2) = \{\{totalDays, lbf, ll, ld\}, \{fbf\}, \{fl\}, \{fd\}\}$
$hDep(a_0, e_1) = \{\{dest, cur\}, \{amount\}\}$
$hDep(a_2, e_3) = \{\}$
$aDep(a_0, e_1, a_2) = true$
$aDep(a_2, e_3, a_2) = false$

(c) dependency sets for `NewReportActivity` and `ItemizedReportActivity`.

#	$\mathbf{WC_{(dest,cur)}}$	$\mathbf{WC_{(amount)}}$
1	Rome, Euro	100
2	Rome, Dollar	0
3	Rome, Pound	
4	London, Euro	
5	London, Dollar	
.	.	
.	.	
30	Berlin, Pound	

(d) widget combinations for dependent widgets in a_0 with respect to e_1.

#	$\mathbf{WC_{hDep(a_0,e_1)}}$
1	Rome, Euro, 100
2	Rome, Dollar, 0
3	Rome, Pound, 100
4	London, Euro, 0
5	London, Dollar, 100
.	.
.	.
30	Berlin, Pound, 100

(e) final set of combinations for Activity a_0 (`NewReportActivity`) in the context of event handler e_1.

#	$\mathbf{AT_{hDep(a_0,e_1)}}$
1	{{Rome , Euro, 100}, ItemizedReport}
2	{{Rome , Dollar, 0}, ItemizedReport}
3	{{Rome , Pound, 100}, ItemizedReport}
4	{{London , Euro, 0}, ItemizedReport}
5	{{London , Dollar, 100}, ItemizedReport}
.	.
.	.
30	{{Berlin, Pound, 0}, ItemizedReport}

(f) generation of Activity Tests for a_0 with respect to e_1.

Figure 6.2: An example to illustrate TrimDroid's generation of tests.

dependencies (recall Algorithm 4) are all denoted in Figure 6.2c.

6.4.1 Activity Test Generation

TrimDroid generate tests for an Activity a in three steps:

(Step 1) For each event $e \in eHandlers(a)$, TrimDroid uses Alloy Analyzer to enumerate over all combinations of widget values in a that are dependent on e. To determine those combinations, TrimDroid utilizes the set of handler dependencies. Let $h \in hDep(a, e)$ represent a set of dependent widgets with respect to an event handler e. TrimDroid calculates WC_h, i.e., the widget combinations for h, using the Alloy Analyzer, as the Cartesian product of all the input classes for its widgets:

$$WC_h \equiv \bigotimes_{j=1}^{|h|} ic(w_j), where\ w_j \in h$$

For instance, in ERS as shown in Figure 6.2c, we can see that $hDep(a_0, e_1)$ is comprised of two sets: $\{dest, cur\}$ and $\{amount\}$. Each one of these two sets indicates a widget dependency among its members, as well as a handler dependency with respect to e_1 (i.e., `"ItemizedReport.onClick"`). We can determine the combination of their elements as shown in Figure 6.2d.

(Step 2) To generate tests for Activity a with respect to event e, every widget $w \in h$, where $h \in hDep(a, e)$, must be assigned a value. To achieve this, all widget combinations, i.e., all instances of WC_h, are combined into one final set. However, since these widgets are independent from one another, I simply need to *merge* them, rather than calculate their cross-product, as follows:

$$WC_{hDep(a,e)} \equiv merge(H), where\ H = \{h | h \in hDep(a, e)\}$$

Definition 3 (Merge)**.** Given a set of sets S, let

$m = \forall s \in S, max(|s|)$, I merge all members of S into C defined as follows:

$$C = \{c_i | c_i \equiv \bigcup_{\forall s \in S} x_{(i \mod |s|)},$$

$$where\ 0 \leq i \leq (m-1) \wedge x_i \in s\}$$

Thus, considering the widget combination in Figure 6.2d, I can calculate the final set of combinations for Activity a_0 (`NewReportActivity`) in the context of event handler e_1 (`ItemizedReport.onClick`) as shown in Figure 6.2e.

As shown in Figure 6.2a, $|ic(dest)| = 10$, $|ic(cur)| = 3$ and $|ic(amount)| = 2$. Thus, merging $WC_{\{dest,cur\}}$ with $WC_{\{amount\}}$ produces 30 unique possible combinations. Note that since $WC_{\{amount\}}$ only has 2 combinations, when I merge it with $WC_{\{dest,cur\}}$, which has 30 combinations, I simply need to ensure all of its unique values are included in the generated tests. In this case, I chose values 100 and 0 for the first two combinations and simply chose 100 for all the remaining combinations. Since I know that *amount* does not interact with the other widgets, I just need to ensure all of its unique values are included in the combinations. However, I still need to include a value for *amount* for all combinations, as the event handler depends on it.

(Step 3) Given all widget value combinations for an Activity a in relation to an event handler $e \in eHandlers(a)$, I can now construct all of the corresponding activity tests $AT_{hDep(a,e)}$. To that end, I simply augment each element of the set $WC_{hDep(a,e)}$ with the action corresponding to the triggering of event handler, i.e., e, as follows:

$$AT_{hDep(a,e)} \equiv WC_{hDep(a,e)} \bigotimes e$$

For instance, in the running example, the set of test combinations for Activity a_0 (`NewReportActivity`) in relation to e_1 (`ItemizedReport.onClick`) is represented in Figure 6.2f. On the other hand, to test Activity a_2 (`ItemizedReportActivity`) in relation to

e_3 (`Reset.onClick`) no value combinations are needed, as $WC_{hDep(a_2,e_3)} = \{\}$.

Tests for an Activity a can be calculated as the union of all generated tests in relation to its event handlers:

$$AT_a \equiv \bigcup_{\forall e \in eHandlers(a)} AT_{hDep(a,e)}$$

This section illustrated two ways in which TrimDroid reduces the number of generated tests. First, since TrimDroid has determined that $\{dest, cur\}$ is independent of $\{amount\}$, instead of calculating all their combinations by taking their cross-product, it simply merges the two sets of combinations (recall Definition 3). Second, since TrimDroid detects there are no dependencies between the *"Reset"* button's event handler and any of the widgets on `ItemizedReportActivity`, it does not generate tests involving any of the widget combinations for that particular event.

6.4.2 System Test Generation

To generate the GUI system tests ST for a given path $a_i \xrightarrow{e_p} a_j \xrightarrow{e_q} a_k$ in an ATM, I first generate the activity tests for each transition (event handler) in the manner described in the previous section. Next, if a_i and a_j are dependent with respect to e_p (i.e., $aDep(a_i, e_p, a_j) = true$), I enumerate all combinations of $AT_{hDep(a_i,e_p)}$ and $AT_{hDep(a_j,e_q)}$ by calculating their cross-product:

$$ST_{a_i \xrightarrow{e_p} a_j \xrightarrow{e_q}} \equiv AT_{hDep(a_i,e_p)} \bigotimes AT_{hDep(a_j,e_q)}$$

Otherwise, a_i and a_j are independent with respect to e_p, in which case I apply the merge operator, resulting in a reduction of generated tests:

$$ST_{a_i \xrightarrow{e_p} a_j \xrightarrow{e_q}} \equiv merge(AT_{hDep(a_i,e_p)}, AT_{hDep(a_j,e_q)})$$

#	$\mathbf{ST}_{\mathbf{a_0 \xrightarrow{e_1} a_2 \xrightarrow{e_3} a_2}}$
1	$\langle\{\{Rome, Euro, 100\}, ItemizedReport\}, \{Reset\}\rangle$
2	$\langle\{\{Rome, Dollar, 100\}, ItemizedReport\}, \{Reset\}\rangle$
3	$\langle\{\{Rome, Pound, 100\}, ItemizedReport\}, \{Reset\}\rangle$
4	$\langle\{\{London, Euro, 100\}, ItemizedReport\}, \{Reset\}\rangle$
5	$\langle\{\{London, Dollar, 100\}, ItemizedReport\}, \{Reset\}\rangle$
.	.
.	.
30	$\langle\{\{Berlin, Pound, 100\}, ItemizedReport\}, \{Reset\}\rangle$

Figure 6.3: System tests for the path $a_0 \xrightarrow{e_1} a_2 \xrightarrow{e_3} a_2$ in ERS

Note that for transition $a_j \xrightarrow{e_q} a_k$ that ends with a final Activity a_k (e.g., a_4 in Figure 4.1), a_k has no activity tests as it has no outgoing transition, and in turn, contributes no combinations to the system tests.

To illustrate the process, consider a situation in ERS where the goal is to generate a system test for the path $a_0 \xrightarrow{e_1} a_2 \xrightarrow{e_3} a_2$ in Figure 4.1. There is an activity dependency between a_0 and a_2 with respect to e_1, i.e., $aDep(a_0, e_1, a_2) = true$ as shown in Figure 6.2c. I thus create all combinations of activity tests for both activities in that transition to build the system tests ST, as shown in Figure 6.3.

My approach produces a total of 30 system tests for this path, each indicated as a sequence in the set ST. Following the generation of system tests, *Test-Case Generation* transforms each test case to proper Robotium format for execution [31].

6.5 Evaluation

To evaluate my approach, I measure TrimDroid's ability to reduce test suites, while maintaining effectiveness. To assess effectiveness, I compare TrimDroid's code coverage and execution time against exhaustive combinatorial testing as well as prior Android testing techniques. Specifically, I investigate the following three research questions:

- **RQ1:** How do TrimDroid, exhaustive GUI combinatorial, and pairwise testing compare with respect to the size of generated test suites and their execution time?
- **RQ2:** How do TrimDroid, exhaustive GUI combinatorial, and pairwise testing compare with respect to code coverage?
- **RQ3:** How effective is TrimDroid compared to prior Android test automation techniques?

For investigating these questions, I use several real-world apps from an open-source repository, called F-Droid [89]. Each selected app satisfies the following criteria: (1) its source code is available; and (2) it uses only standard GUI widgets of the Android Application Development Framework, e.g., it is a *native mobile app*, rather than a *web mobile app*. The first criterion ensures that I can properly measure code coverage; the second criterion is due to the limitation of TrimDroid's static program analysis that only supports standard Android libraries and widgets. These apps are selected from different categories, such as productivity, entertainment, and tools.

Table 6.1 lists these apps. For each *App*, Table 6.1 depicts its size as measured using lines of code (*LOC*). I compare TrimDroid's coverage against exhaustive combinatorial testing, since its coverage *subsumes* the coverage of all other combinatorial testing techniques [20]. I used prime path coverage criterion (recall Section 6.3) for both exhaustive testing and TrimDroid to allow for a fair comparison.

I also compare TrimDroid's coverage with M[agi]C [19]—a pairwise GUI combinatorial testing technique that has been applied on Android apps, among others. M[agi]C requires the user to manually construct two types of models for the software under test: a model identifying the input classes for all the widgets, and a model that captures the transitions between the screens. In fact, the former is equivalent to TrimDroid's *ATM*, and the latter to its *IM*. To ensure a fair comparison, I manually transformed the *ATM* and *IM* models that TrimDroid generated automatically for each app into models that can be used by M[agi]C. M[agi]C uses a post-optimization algorithm that reduces the number of generated test cases after executing them once. This is achieved by removing the input combinations for paths that share events. Finally, I did not use the optimization option of M[agi]C to ensure the generated test cases achieve the maximum code coverage.

Although assessing TrimDroid's fault-detection ability is a primary concern of ours, currently there is no organized set of open-source Android apps with known defects and fault reports that can be used to evaluate TrimDroid's fault detection ability. Alternatively, mutation testing can be used, where the mutants replicate actual faults. Unfortunately, there is no support for mutation testing of Android apps to this date. Particularly, no fault model exists for Android apps, preventing production of mutants that can substitute for real faults. One of the authors has recently begun to investigate the challenges of mutation testing for Android applications [95, 96].

All of my experiments were conducted on a machine with 16GB memory and a quad core 2.3GHz processor. I used Android Virtual Devices (Android emulators) with 2GB RAM, 1GB SD Card, and the latest version of Android that is compatible with the app, except for Dynodroid, whose in-box emulator uses Android 2.3. A fresh emulator was created for each app along with only default system applications. During the experiments, I used EMMA [88] to monitor code coverage. Specifically, I measured line coverage by running all of the generated test cases on each app. TrimDroid, subject apps, and my research artifacts are publicly available [97].

6.5.1 Test-Suite Reduction

To answer RQ1, I compare the test suites generated by TrimDroid, exhaustive combinatorial testing and M[agi]C in terms of size and execution time. For each *App*, Table 6.1 shows the size and execution time of test cases for both techniques. The table also shows the reduction of test cases compared to exhaustive combinatorial testing in the right-most column.

I observe that in most cases TrimDroid is able to significantly reduce the number of generated tests compared to exhaustive testing. TrimDroid, on average, generates 57.86% fewer tests compared to exhaustive testing. The smaller number of tests that would need

Table 6.1: Pruning effect in TrimDroid

App	**LOC**	**Exhaustive Testing**		**M[agi]C**		**TrimDroid**		**Reduction**
		Test Cases	Time(s)	Test Cases	Time(s)	Test Cases	Time(s)	
HashPass	429	128	515	15	50	32	126	75.00%
Tipster	423	36	243	20	44	24	156	33.33%
MunchLife	631	10	84	9	37	8	56	20%
Blinkenlights	851	54	252	5	36	22	112	59.25%
JustSit	849	50	236	10	42	16	74	68%
autoanswer	999	576	5655	17	196	12	118	97.91%
AnyCut	1095	6	38	4	38	6	38	0%
DoF Calculator	1321	1174	7292	107	953	30	373	97.44%
Divide&Conquer	1824	12	85	5	47	4	32	66.66%
PasswordGene	2824	> 48000	–	58	351	418	1263	99.91%
TippyTipper	2953	26	238	30	172	25	225	3.84%
androidtoken	3680	454	13336	19	189	42	686	90.74%
httpMon	4939	42	407	34	235	28	282	33.33%
Remembeer	5915	48	633	24	194	17	320	64.58%

to be inspected, especially for human engineers, would result in significant savings in time and effort. By doing so, TrimDroid reduces the time needed to execute the tests by 57.46% on average. The savings are more pronounced in certain cases. For PasswordGenerator, TrimDroid eliminates more than 479,000 tests, which is a reduction in tests by multiple orders of magnitude. Furthermore, exhaustive testing crashes for PasswordGenerator (as denoted by the dash in Table 6.1) before generating all the app's test cases, as the massive size of its generated test suite depletes the machine's memory.

On average, TrimDroid generates 2 times more test cases than M[agi]C. However, as described later, the tests generated by TrimDroid achieve a substantially higher code coverage. Recall that while TrimDroid adopts t-way testing, where t is determined according to the dependencies extracted through program analysis, M[agi]C uses a fixed t for all apps, i.e., two. Apps for which TrimDroid has produced more tests than M[agi]C, harbor complex dependencies involving three or more widgets. Apps for which TrimDroid has produced fewer tests are lacking dependencies among their widgets, indicating that the pairwise strategy is producing unnecessary tests.

6.5.2 Effectiveness - Exhaustive and Pairwise

To answer RQ2, I compare the statement coverage resulting from the execution of test suites generated by TrimDroid, M[agi]C and exhaustive testing. The results are summarized in Figure 6.4. Each application is identified along the horizontal axis, while the vertical axis shows the statement coverage achieved by TrimDroid, M[agi]C and exhaustive testing. In all cases, TrimDroid achieves at least the same statement coverage as exhaustive testing and the same or better statement coverage than M[agi]C. For the *PasswordGenerator* app, exhaustive testing's failure to complete is depicted as 0% coverage, since the inability to generate the test suite is effectively 0% statement coverage.

Note that some subject apps (e.g., *autoanswer* and *httpmon*) heavily use Service components. Unlike Activity components, Service components are responsible for handling events initiated by the system rather than the GUI. For example, *autoanswer* provides Services that perform a task based on a set of predefined preferences when a phone call is received. Given that TrimDroid's focus is on GUI testing, it is no surprise that it does not achieve good coverage for these types of apps. In fact, when I compare against the highest possible coverage for a GUI-based testing approach in such apps, namely exhaustive GUI testing, we observe that TrimDroid achieves the same coverage.

Thus, in comparison to exhaustive testing, the results show that, although TrimDroid significantly reduces the number of tests, and subsequently their execution time, the resulting code coverage is not degraded at all. In principle, however, due to the limitations of static analysis (e.g., unsupported Android libraries), it is possible for TrimDroid to achieve less coverage than exhaustive testing, even though my experiments have not yet revealed such instances.

On average, TrimDroid achieves 13% more statement coverage than M[agi]C. This result supports the effectiveness of using the proposed dependency-based heuristics for reducing the number of tests, rather than fixed strategies, such as pairwise testing, that compromise on coverage.

6.5.3 Effectiveness - Other Android Testing

A meaningful comparison of Android test automation techniques is generally difficult, as each has its own unique objective. My objective in TrimDroid has been to *reduce the number of tests in combinatorial GUI testing of Android apps without compromising on coverage.* On the other hand, several prior techniques have aimed to maximize code coverage through search-based techniques, regardless of the number of tests it takes to do so, which could pose a significant burden when the assessment of whether the tests have passed or failed entails manual effort. Nevertheless, I compare against the code coverage and execution time achieved by four prior techniques: Monkey [37], Dynodroid [8], M[agi]C [19] and my prior work, EvoDroid [47].

Android Monkey, a widely used testing technique developed by Google, represents the state-of-practice and operates by sending random inputs and events to the app under test.

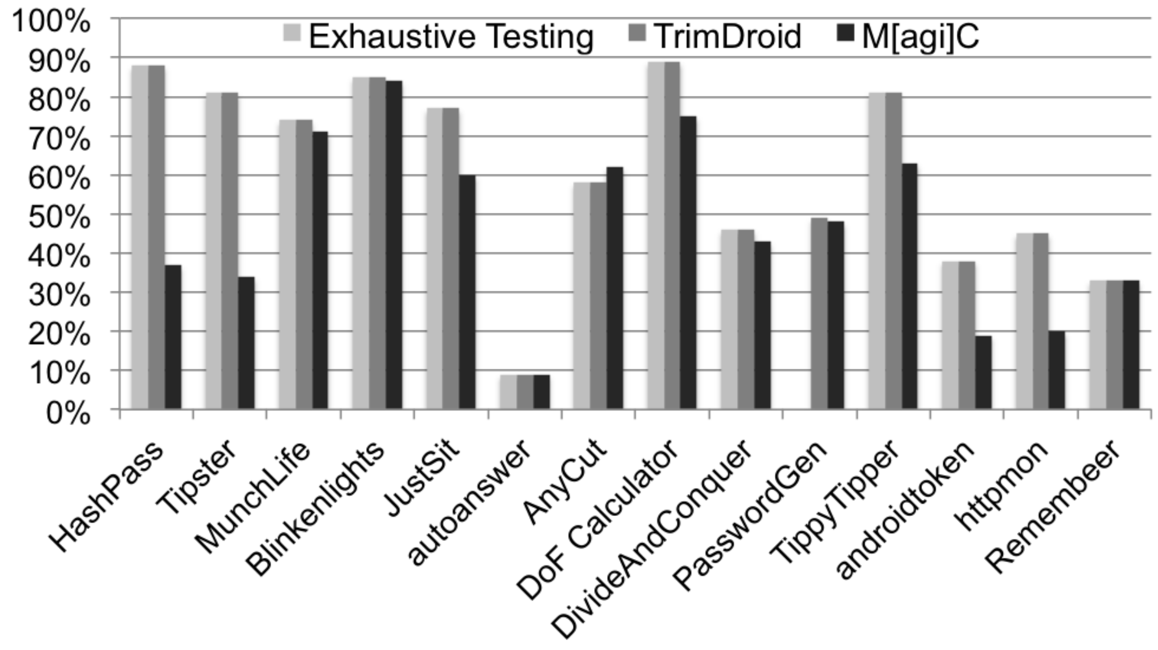

Figure 6.4: Statement coverage comparison

Dynodroid uses several heuristics to improve on the number of inputs/events used by Monkey, and thus achieves similar coverage with fewer generated events. As both Monkey and Dynodroid are based on pseudo-random testing, using the same low number of events that are generated by TrimDroid may not be a fair comparison. To address that, I ran both Dynodroid and Monkey with 2,000 input events, which is the maximum input size for Dynodroid [8].

EvoDroid is a system testing technique that implements a novel evolutionary testing algorithm. EvoDroid's fitness is designed to maximize the statement coverage through exhaustively exploring the search space for event sequences. Note that EvoDroid is a search-based testing technique; thus, using the same low number of events that are generated by TrimDroid is not adequate for the evolutionary search to be effective. On the other hand, the main goal of TrimDroid is to generate a limited number of tests, which is crucial when the evaluation of tests (i.e., oracle) involves manual effort. To summarize, EvoDroid is intended to exhaustively test event sequences; TrimDroid is designed to comprehensively

Table 6.2: Comparison of TrimDroid with other techniques

App	**TrimDroid**		**M[agi]C**		**Monkey**		**Dynodroid**		**EvoDroid**	
	Coverage	Time(s)	Coverage	Time(s)	Coverage	Time(s)	Coverage	Time(s)	Coverage	Time(s)
HashPass	88%	126	37%	50	40%	41	57%	33917	57%	23472
Tipster	81%	156	34%	44	67%	104	59%	33825	89%	22813
MunchLife	74%	56	71%	37	49%	75	54%	31421	78%	20965
Blinkenlights	85%	112	84%	36	49%	49	81%	25278	63%	22418
JustSit	77%	74	60%	42	35%	163	53%	41252	84%	20391
autoanswer	9%	118	9%	196	6%	43	10%	59672	8%	21883
AnyCut	58%	38	62%	38	6%	71	66%	21757	84%	19735
DoF Calculator	89%	373	75%	953	43%	70	-	-	36%	20713
DivideAndConquer	46%	32	43%	37	51%	58	83%	33644	-	-
PasswordGen	49%	1263	48%	351	42%	128	33%	31882	62%	26129
TippyTipper	81%	225	63%	172	42%	106	-	-	82%	23108
androidtoken	38%	686	19%	189	10%	67	11%	57813	29%	19188
httpmon	45%	282	20%	235	4%	46	6%	22563	36%	23403
Remembeer	33%	320	33%	194	27%	46	23%	53013	28%	22517

test the input space of GUI widgets, using a limited number of event sequences identified by utilizing prime paths. Consequently, a fair one-to-one comparison of the two techniques might not be possible. Having said that, I ran EvoDroid for ten evolutionary generations on all apps, which is the same setup as that used in [47], and compare the resulting statement coverage.

Few other tools exist, but I were unable to include them in my experiments, as I could not properly run them after significant consultation with their developers. A^3E [41] aims to discover the Activities comprising an app by covering a model similar to the notion of *ATM*. I were unable to run A^3E on any of my apps using the virtual machine provided by its developers. A^3E gets stuck when trying to start Troyd [98]—an integration testing framework for Android utilized by the tool.

I also attempted to run *SwiftHand* [5]. It uses (1) machine learning to infer a model of the app during testing, (2) the inferred model to generate user inputs that visit unexplored states of the app, and (3) the execution of the app on the generated inputs to refine the model. SwiftHand exits with an exception failing to locate the main Activity of the app. Based on my analysis, the issue may reside with the custom made instrumentation of the app under test. My attempts to resolve the issues with the help of the tool developers have been unsuccessful to date.

The statement coverage and execution time for all five testing techniques are summarized in Table 6.2. Dynodroid cannot run on *TippyTipper* and *DoF Calculator*—denoted by a dash (-) in Table 6.2—since the newer Android APIs utilized by those apps are not supported by Dynodroid. I could not run the current version of EvoDroid on *DivideAndConquer* due to use of unsupported APIs.

The results show that TrimDroid is able to achieve higher code coverage in most cases. Note that TrimDroid is targeted at GUI testing, and therefore only generates GUI events, while Dynodroid and EvoDroid support both system events as well as GUI events. As a result, for some apps, TrimDroid cannot achieve the same coverage as Dynodroid and EvoDroid. Nevertheless, TrimDroid's coverage, on average, outperforms Monkey by 27.36%, Dynodroid by 16.33%, M[agi]C by 14%, and EvoDroid by 4%.

In addition, TrimDroid runs 134 times faster than Dynodroid, 78 times faster than EvoDroid, 1.5 times slower than M[agi]C, and about 3 times slower than Monkey. TrimDroid's slower performance in comparison to M[agi]C and Monkey can be attributed to the analysis performed for extracting the models as well as the application of heuristics for reducing the number of tests.

6.5.4 The Curious Case of Duplicate Test-cases

Although Alloy constraint solver employs a heuristic algorithm to eliminate as many isomorphic solutions as possible, removing isomorphic solutions is essentially an NP-hard problem [21]. As a result, Alloy does not guarantee that the produced solution set is minimal and does not contain any duplicates [99]. To deal with this issue TrimDroid employs a mechanism to filter-out the redundant test-cases by identifying the duplicate paths and combination of values from the SAT solver results.

The statement coverage and execution time for test-cases that contain no duplicates are summarized in Table 6.3. The results illustrate that although removing the duplicates will result into less number of test cases for both exhaustive approach and TrimDroid, the average reduction in the number of test cases remains almost the same.

6.6 Conclusion

This chapter presented a fully-automated approach for generating GUI system tests for Android apps using a novel combinatorial technique. This approach employs program analysis to partition the GUI input widgets into sets of dependent widgets. Thus, the GUI widgets with dependencies become candidates for combinatorial testing. Then, an efficient constraint solver is used to enumerate the test cases covering all possible combinations of dependent GUI widgets. The experimental evaluation shows that TrimDroid is able to significantly reduce the number of tests in comparison to exhaustive combinatorial testing, without any degradation in code coverage.

Recall from Section 6.4 that the current implementation of TrimDroid uses a predefined set of input classes for unbounded widgets. The next chapter, provides the details for an extension of TrimDroid that uses symbolic evaluation to systematically derive the input classes for those widgets.

Table 6.3: TrimDroid results after removing the duplicate test-cases

App	**Exhaustive Testing**			**TrimDroid**			**Reduction**
	Test cases	Coverage	Time(s)	Test cases	Coverage	Time(s)	
HashPass	64	88%	227	16	88%	63	75.00%
Tipster	15	81%	95	9	81%	59	40.00%
MunchLife	10	74%	78	8	74%	54	20.00%
Blinkenlights	12	85%	58	6	85%	38	50.00%
JustSit	22	77%	112	8	77%	45	63.63%
autoanswer	288	9%	2883	6	9%	59	97.91%
AnyCut	6	58%	46	6	58%	46	0%
DoF Calculator	587	89%	3603	15	89%	154	97.44%
DivideAndConquer	2	46%	32	2	46%	32	0%
PasswordGen	> 480000	-	-	19	49%	118	99.99%
TippyTipper	26	81%	174	25	81%	165	3.84%
androidtoken	118	38%	1472	15	38%	210	87.28%
httpmon	42	45%	447	29	45%	281	30.95%
Remembeer	48	33%	592	17	33%	191	64.58%

Chapter 7: Systematic Input Partitioning

It can be said that the main goal of test input generation is to select the best candidates from the domain of all possible values for an input parameter. As shown in the previous chapter, if the input domain for a parameter is bounded and contains a relatively small number of values, all values can be considered as candidates for test combinations. But in the case of unbounded domains, such as real numbers or, even very large bounded domains, such as integer numbers, the input space is quite large. Hence, considering all the values in those domains is technically impossible in practice. Traditionally, combinatorial testing approaches assume that the input classes for each input parameter are defined in either the software specification or manually by the human tester [15]. In practice, most Android apps lack such detailed specification. On the other hand, manually defined domains are both imprecise and hard to maintain as they have to be kept updated with any changes in the app itself.

Several techniques have been proposed to target the problem of input domain partitioning. *Equivalence partitioning* breaks down the input domain into equivalent partitions by either using the software specifications or certain heuristics [56]. *Boundary value analysis* only considers the boundary values as test inputs [100]. Finally, *random sampling* proposes selecting random values based on a statistical distribution model for each domain [4].

Input space partitioning suggests that each input domain can be partitioned into classes that are assumed to contain equally useful values from a testing perspective. As a result, a smaller collection of values that produces the same results as the whole domain would be just as good. For example, in the ERS app, the input domain for `amount` can be broken down into two classes, where `amount > 500` and `amount < 500`. In this case, only two values (i.e., 1 and 501) are needed to represent the whole input domain. Hence, selecting values that are representative of each partition of the input domain results in more relevant combinations for testing the software.

As mentioned, input domains are broken down into classes using some sort of information about a program, e.g., documented specifications on features of the code, or the knowledge of a programmer about the code. However, to the best of my knowledge, no prior research has explored a fully automated program analysis approach to extract the input classes for the input parameters of a program. This is mainly due to the known inefficiencies of program analysis techniques, such as symbolic execution especially in dealing with real-world programs [4].

Mahmood et al. [47] show that Android applications can be sliced into separate segments each of which can be used a chromosome for evolutionary testing of Android apps. The same concept of segments, can be used to slice the app into smaller parts that are targeted by program analysis. In this case, the required program analysis is limited to an Activity and is not context sensitive. As a result, there would be no need to analyze the external Android libraries and the program analysis would be able to support real apps without any additional effort.

This chapter presents a light weight program analysis technique that is used along with an SMT solver to produce the input classes for unbounded input widgets automatically. The extracted input classes are added to the IM and then used by TrimDroid to generate the test inputs. The experiments corroborate that this approach improves the statement coverage for TrimDroid. Hence, it can be used as a viable substitute for symbolic execution for testing Android application.

7.1 Approach

Figure 7.1 depicts a high-level overview of the process for extracting the input classes. This process is comprised of three major step. First is to identify all unbounded data widgets

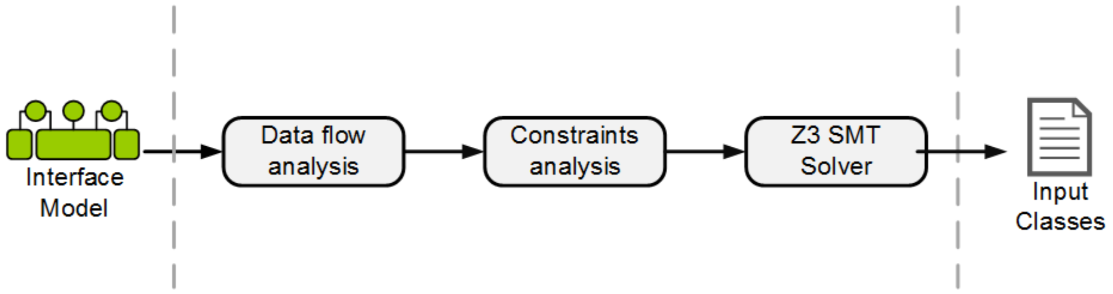

Figure 7.1: High-level overview of input class extraction.

using the app's *IM* and perform data-flow analysis to extract the *definition-use chain* [1] of each widget. Next, the constraints around all variables in the widget's definition-use chain are collected. Finally, an efficient SMT solver it used to enumerate the result values for each constraint. Theses values are used to define the input classes in TrimDroid.

In order to determine the input classes for unbounded data widgets, I first use the *IM* to identify such widgets in each activity of the app. This is followed by performing data-flow analysis to explore the definition-use chain of each widget. The definition-use chain information of each widget is stored in appropriate data structures in a way that each reference, r, can be easily associated to the widget it uses.

An example of this is shown in Listing 7.1, where *R.id.amount* represents the id for the `amount` text-box on `NewReportActivity`. In this case, the data-flow analysis will determine the set of references that use the *amount* widget as {`amount`, `amountValue`, `total`} :

The next step is to explore the code and extract the constraints for every unbounded widget of each activity. Algorithm 5 defines the process of extracting the constraints for references to unbounded widgets. The algorithm takes an Activity a as input and produces $pairs(w, C)$, a map of unbounded widgets w to a set of constraints C.

For each method, the constraint extraction algorithm iterates over each reference r that is used in a conditional statement and determines if r refers to a widget (lines 2–4 of

[1]Definition-Use Chain is a data structure consisting of a definition of a variable and all the uses reachable from that definition

```
public void onClick (View v) {
  ...
  EditText amount = (EditText)findViewById(R.id.amountId);
  int amountValue = Integer.parseInt(amount.getText().toString());
  if (amountValue <= 500) {
          ...
          int total = amountValue + totalDays * dailyAllowance;
  }
  ...
}
```

Listing 7.1: sample code code snippet from `NewReportActivity` in ERS.

Algorithm 5). To make that determination, $isAWidget(r)$ traverses the definition-use chain of r to determine if any of its definitions refers to a widget. At this point, the widget id w associated with the reference r is acquired (line 5). Finally, the algorithm uses the *IM* to check if widget w is an unbounded widget (recall from Chapter 4 that the *IM* extracts and stores the information about every widget of each activity such as name, type, and etc. from the layout files). In that case the constraint of the conditional statement c is added to the map of constraints associated with w i.e., $pairs(w, C)$ (lines 6–8 of Algorithm 5).

For the the sample code in Listing 7.1, the constraint extraction algorithm iterates over all conditional statements of `onClick()` method. Then, it tries to associate each reference used in conditional statements (i.e., `amountValue`) to an unbounded widget, i.e., `R.id.amoundId`. Finally, the constraint for the widget is added to the result map i.e., {`R.id.amoundId`, { `amountValue <= 500` } }.

The data-flow analysis and the constraints analysis components are implemented on top of the Soot framework [85]. Although Soot framework was initially created as a Java compiler test-bed it has advanced to support analysis of Java bytecode. Furthermore, Soot has been integrated with the *Dexpler* transformer [86] to translate Android's dalvik bytecode into the Soot's intermediate representation language, called *Jimple*. Currently, Soot supports major library classes of Android (such as *Intent*, *Context*, *Application* and

Algorithm 5: Constraint Extraction

```
   Input: $a \in A$
   Output: $pairs(w, C)$ where $w \in widgets(a)$ and $C \subset constraints(a))$
1  $pairs(w, C) \leftarrow \emptyset$;
2  foreach $meth \in a.Methods$ do
3      foreach $r$ used in a conditional statement of $meth$ with constraint $c$ do
4          if $isAWidget(r)$ then
5              $w \leftarrow getWidget(r)$ ;
6              if $isUnbounded(w)$ then
7                  $pairs(w, C) \leftarrow pairs(w, C \cup c)$;
8                  break;
```

so on) and creates a *phantom class*[2] for each reference that it cannot resolve.

Finally, I use an off-the-shelf SMT solver, Z3 [57], to compute the test inputs for satisfiable constraints for each widget. For each unbounded widget, the generated input classes are added to the *IM* to be used by TrimDroid. These values are transformed to Alloy representations in the same fashion as described in Chapter 6 and are used by TrimDroid for generating the test combinations.

The main difference between this approach and symbolic execution is that here the constraints are considered individually, whereas in symbolic execution each constraint is a part of a path-condition. As a result, in this approach the SMT solver only deals with simple individual constraints instead of complex path-conditions. Accordingly, the acquired values from each constraint are used as input classes for widgets in combinatorial testing.

To illustrate this approach, consider the Java program shown in Figure 2.4a, where S0, S1, S2, S3, and S4 denote statements that can be invoked in different paths of the program. As explained in Chapter 2, symbolic execution solves all the paths in the tree resulting in the pairs shown it Table 7.2a.

Although in theory symbolic execution can produce the minimal set of values for exploring all execution paths of a program, as mentioned in Chapter 5 symbolic execution of real-world programs is not always possible. Recall from Chapter 5 that current SMT

[2]Phantom references are the weakest level of reference in Java [101].

Path Condition	**Generated Value**
$!(X > 0)$	$X = 0$
$X > 0$ && $Y == 5$	$X = 1, Y = 5$
$!(X > 0$ && $Y == 5)$	$X = 1, Y = 6$
$X > 3$ && $Y < 10$	$X = 4, Y = 11$
$!(X > 3$ && $Y < 10)$	$X = 3, Y = 10$

(a)

Individual Constraint	**Generated Value**
$x > 0$	$X = 1, X = -1$
$y == 5$	$Y = 5, Y = 6$
$x > 3$	$X = 4, X = 3$
$y > 10$	$Y = 11, Y = 10$

(b)

Figure 7.2: Values for X and Y generated by (a) symbolic execution, (b) constraint solving and combinatorial testing

solvers fail to solve complex path-conditions. Moreover, symbolic execution relies on a more complicated inter-procedural program analysis for constructing the path-conditions. As a result, developing a symbolic execution approach that can be applied to a large set of Android apps would require a lot of engineering effort to model the Android libraries that the apps depend on.

To address the above issues, we can solve the constraints individually (as shown in Figure 7.2b) to partition the input domains of X and Y independent of each other. In that case we end up with four different values for $IC(X)$ and four different values for $IC(Y)$. The cross-product of the $IC(X)$ and $IC(Y)$ will generate 16 combination of values which result in the execution of same branches. Although, this approach results in more test inputs for exercising all execution paths of an app compared to symbolic execution (i.e., 16 instead of 5), it uses a more efficient and scalable program analysis technique than symbolic execution.

7.2 Evaluation

To assess effectiveness of using the extracted input classes for unbounded data widgets, I measure improvements in TrimDroid's statement coverage as well as the overhead of performing the additional analysis. Specifically, I investigate the following two research questions:

- **RQ1:** How do manually defined and automatically extracted input classes affect TrimDroid's statement coverage?
- **RQ2:** How does extracting the input classes automatically affect the performance of TrimDroid?

For investigating the above research questions, I use the same real-world apps from F-Droid [89] that are used to evaluate TrimDroid in Chapter 6. All of my experiments were conducted using the same setup as the experiments in Chapter 6. A fresh emulator was created for each app along with only default system applications. I used EMMA [88] to monitor statement coverage for this experiment.

7.2.1 Effect on Statement Coverage

To investigate RQ1, I compare TrimDroid's coverage in two cases: (1) using two manually defined values for unbounded widgets (i.e., 0, 1000) and (2) using automatically extracted input classes.

The results show that in apps where conditional statements are defined on top of unbounded GUI widgets (i.e., Tipster and DoF Calculator), the statement code coverage is improved. At the same time, the results support the findings in the empirical study in Chapter 5 that many Android apps at the moment do not have complicated constraints on the GUI widgets. Further study of these apps showed that even in apps that have some conditions on the input values, the conditional blocks are relatively small compared to the rest of the source-code in terms of lines of code (only few lines that are in the block). Hence, the added coverage is not that significant. In future, as Android devices would become more computationally powerful and apps grow in complexity, I expect the technique described here to have a more substantial impact on the results.

Table 7.1: TrimDroid results after adding the input classes for unbounded widgets

App	**TrimDroid**		**Exhaustive Testing with input classes**		**TrimDroid with input classes**		
	Test cases	Coverage	Test cases	Coverage	Test cases	Coverage	Overhead(s)
HashPass	16	88%	64	88%	16	88%	3
Tipster	9	81%	30	88%	9	88%	9
MunchLife	8	74%	10	74%	8	74%	4
Blinkenlights	6	85%	12	85%	6	85%	2
JustSit	8	77%	22	77%	8	77%	3
autoanswer	6	9%	288	9%	6	9%	2
AnyCut	6	55%	6	58%	6	58%	2
DoF Calculator	15	89%	1174	89%	15	93%	12
DivideAndConquer	2	46%	2	46%	2	46%	2
PasswordGen	19	49%	> 480000	-	19	49%	1
TippyTipper	25	81%	26	81%	25	81%	5
androidtoken	15	38%	118	38%	15	38%	6
httpmon	29	45%	42	45%	29	45%	3
Remembeer	17	33%	48	33%	17	33%	4

7.2.2 Effect on Efficiency

To answer RQ2, I capture the program analysis overhead for extracting the input classes. To calculate this overhead, I log the time stamp right before calling the input class extraction component and after the extraction process finishes. Table 7.1 illustrates the results from this experiment. The results show that the overhead is insignificant for all subjects.

7.3 Conclusion

The success of combinatorial testing depends on how well the domain spaces for input parameters of the system under test are partitioned [15]. As mentioned in Chapter 6, TrimDroid uses the layout XML files to define the input classes for bounded widgets such as check-boxes, radio buttons and so on.

This chapter presented a systematic approach for partitioning the input domain of unbounded GUI widgets in Android apps. This approach employs a light-weight program analysis technique to extract the constraints for unbounded widgets. It then uses an efficient SMT solver to generate proper values to satisfy those constraints. These values are used by TrimDroid to generate the test combinations for the app.

Unlike symbolic execution, the presented approach does not require excessive time and resources for solving complex path conditions. Moreover, the proposed program analysis does not require any engineering effort for modeling Android API classes. Hence, it can be easily extended to support a vast category of Android applications. The experimental evaluation shows that the proposed approach improves TrimDroid's statement coverage while not enforcing a significant overhead.

Chapter 8: Conclusion

Pervasiveness of smartphones and the vast number of corresponding apps have underlined the need for more practical automated software testing techniques. In spite of the wealth of research that has been focused on either unit or GUI testing of smartphone apps, random testing remains to be the state of practice [36]. Applying automated input-generation techniques such as symbolic execution and combinatorial testing to new environments such as Android has shown to be challenging. This dissertation presents two new techniques, SIG-Droid and TrimDroid, to facilitate the automation of test-input generation for Android applications. Both techniques are backed with automated program analysis to extract app models from the the app's byte-code. They leverage two automatically extracted models: *Interface Model* and *Activity Transition Model*.

SIG-Droid solves the three important problems for symbolic execution of Android apps, (1) Dalvik byte-code (2) event-based nature of Android apps, and (3) path-explosion. It uses the *IM* to find values that an app can receive through its interfaces. Those values are then exchanged with symbolic values to deal with constraints with the help of a symbolic execution engine. The *ATM* is used to drive the apps for symbolic execution and generate sequences of events.

On the other hand, TrimDroid introduces a novel approach for t-way combinatorial testing to address the combinatorial explosion of test cases. TrimDroid is a framework for GUI testing of Android apps that uses a novel strategy to generate tests in a combinatorial, yet scalable fashion. It is backed with automated program analysis and formally rigorous test generation engines. It relies on program analysis to extract formal specifications. These specifications express the app's behavior (i.e., control flow between the various app screens) as well as the GUI elements and their dependencies. The dependencies among the GUI elements comprising the app are used to reduce the number of combinations with the help of a solver.

Finally, a novel approach for combining these two techniques is proposed. In this approach, the main goal is to use the strengths of program analysis and constraint satisfaction to address a known weakness of TrimDroid in dealing with unbounded input parameters. It uses light weight program analysis to identify constraints around the variables associated with unbounded GUI widgets. Consequently, a constraint solver is used to generate the values that represent the input classes for those widgets .

The experiments have corroborated that effectiveness and efficiency of both techniques in terms of maximizing the code coverage with a small number of test-cases. They show SIG-Droid is able to achieve significantly higher code coverage than existing automated testing tools targeted for Android and TrimDroid's ability to achieve a comparable coverage as that possible under exhaustive GUI testing using significantly fewer test cases.

8.1 Research Contributions

This dissertation first elaborates the scope of the research problem and an overview of two approaches to address this problem. Then it defines the related concepts and enumerates the related work. Finally, it provides the details of the work done, along with the respective evaluation results which support the effectiveness of the proposed approach. The following is the concrete list of contributions in this research:

- **Model extraction:** I developed a program analysis technique that automatically builds models of the Android apps. These models capture the event-driven behavior of the apps and are used to derive the sequences of events as well as identifying the GUI widgets.

- **Symbolic execution framework for Android:** I provided solutions to solve the challenges of symbolic analysis for Android framework, and extend Symbolic Pathfinder (SPF) [65] to support Android apps. The symbolic execution engine is used to prune the input domain for the unbounded data-widgets.

- **Reduced combinatorial testing for Android:** I developd a fully-automated approach for generating GUI system tests for Android apps using a novel combinatorial technique. This approach employs program analysis to partition the GUI input widgets into sets of dependent widgets. Thus, the GUI widgets with dependencies become candidates for combinatorial testing.

- **Systematic input domain partitioning:** I proposed an automated approach for systematically partitioning the input domain for unbounded data-widgets into input classes. This approach employs program analysis and constraint satisfaction to produce input values that represent the domain of inputs for widgets. These values are used as input classes for combinatorial testing.

- **Tools and implementation:** I have developed a working implementation of SIG-Droid and TrimDroid that automatically generates executable Robotium[31] test cases that include, among other things, event sequences and data inputs for testing of Android apps

8.2 Limitations and Future Work

Although SIG-Droid has shown to be significantly better than existing tools for automated testing of Android apps, there are several avenues of future research and improvement. Currently, I generate the sequence of events through a depth first search on the *ATM*, which does not guarantee to generate all possible sequence of events. For instance, consider a situation in which a particular execution path is taken only when the same button is clicked several times. Covering such sequences requires the depth-first search algorithm to include loops in its search for all unique sequences of events, the space for which is infinite. In our previous work [47], we have developed an evolutionary testing technique for Android apps to support generation of more complex sequences. I plan to use both techniques in tandem to complement the shortcomings of each. The symbolic execution engine presented in this dissertation will be used to solve the constraints in the code, while the proposed evolutionary algorithm will be used to find sequences that reach a particular location in code.

Moreover, the *Model Extraction* currently depends on a fully static approach to inferred the models. Although, all proposed approaches in this document are decoupled from the mechanism that is used by the *Model Extraction*, the ability to achieve high code coverage is only possible if the inferred models are accurate. In practice, due to the potentially complex interactions with the app (e.g., long click, rotate, press key etc.) and dynamically defined GUIs, it is difficult to extract complete models of the app. To address this issue, I plan to explore other static (e.g. Gator [102]) and dynamic approaches (e.g. dynamic EvoDroid [103]) to improve the accuracy of the extracted models.

While the proposed program analysis heuristics have shown to be quite effective in pruning the test combinations, they have some known limitations. For example, the *IM* is constructed by analyzing the layout XML files statically, which does not handle cases where the Activity views are defined dynamically. In addition, our static analysis is subject to false negatives in certain rare cases, e.g., dependencies due to widget values being stored/read from the SD card, or dependencies occurring through global variables. My future research would involve improving the analysis to support such cases.

The premise of this is that the only available resource for automated testing is an app's APK file. If an app's specification is also available (e.g., in a formal machine-interpretable format), one could investigate the extraction of interactions from the app's specification as well as its implementation for combinatorial test generation. In practice, most apps on the market lack specifications that can be used effectively for automated testing. However, this means that if two widgets should have a dependency according to the intended specification of the software, but the implemented software does not have such a dependency, possibly because the developer failed to correctly realize the specification, TrimDroid would not generate tests to exercise those combinations.

In this work, I have focused on automatic test input generation, rather than automatic *test oracle* construction, resulting in oracles that determine whether test cases pass or fail. Test-oracle construction, especially in an automated way, is a significant challenge that is beyond the scope of this dissertation, but certainly an avenue of future research interests. I believe that it will be necessary to have the user in the loop to generate oracles that assess intended app behaviors. Hence, reducing the number of test cases to be inspected is certainly beneficial. Moreover, the ability to generate tests that can achieve high code coverage has applications beyond testing for functional defects: Energy issues, latent malware, and portability problems are important concerns in the context of mobile devices that are often effectively detected by executing the code.

Bibliography

Bibliography

[1] Android Activity, http://developer.android.com/guide/components/activities.html.

[2] Android Service, http://developer.android.com/guide/components/services.html.

[3] R. Cozza, I. Durand, and A. Gupta, "Market Share: Ultramobiles by Region, OS and Form Factor, 4Q13 and 2013," *Gartner Market Research Report*, February 2014.

[4] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. Mcminn, "An orchestrated survey of methodologies for automated software test case generation," *J. Syst. Softw.*, vol. 86, no. 8, pp. 1978–2001, Aug. 2013.

[5] W. Choi, G. Necula, and K. Sen, "Guided gui testing of android apps with minimal restart and approximate learning," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '13. Indianapolis, Indiana, USA: ACM, 2013, pp. 623–640.

[6] C. Hu and I. Neamtiu, "Automating gui testing for android applications," in *Proceedings of the 6th International Workshop on Automation of Software Test*, ser. AST '11. Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 77–83.

[7] W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated gui-model generation of mobile applications," in *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering*, ser. FASE'13. Rome, Italy: Springer-Verlag, 2013, pp. 250–265.

[8] 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*, ser. ESEC/FSE 2013. Saint Petersburg, Russia: ACM, 2013, pp. 224–234.

[9] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using gui ripping for automated testing of android applications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2012. Essen, Germany: ACM, 2012, pp. 258–261.

[10] M. Shafique and Y. Labiche, "A systematic review of state-based test tools," *Int. J. Softw. Tools Technol. Transf.*, vol. 17, no. 1, pp. 59–76, Feb. 2015.

[11] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, "A survey on model-based testing approaches: A systematic review," in *Proceedings of the 1st ACM*

International Workshop on Empirical Assessment of Software Engineering Languages and Technologies: Held in Conjunction with the 22Nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007, ser. WEASELTech '07. Atlanta, Georgia: ACM, 2007, pp. 31–36.

[12] D. Richard Kuhn and V. Okum, "Pseudo-exhaustive testing for software," in *Proceedings of the 30th Annual IEEE/NASA Software Engineering Workshop*, ser. SEW '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 153–158.

[13] J. C. King, "A new approach to program testing," in *Proceedings of the International Conference on Reliable Software*. Los Angeles, California: ACM, 1975, pp. 228–233.

[14] S. Ganov, C. Killmar, S. Khurshid, and D. E. Perry, "Event listener analysis and symbolic execution for testing gui applications," in *Proceedings of the 11th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering*, ser. ICFEM '09. Rio de Janeiro, Brazil: Springer-Verlag, 2009, pp. 69–87.

[15] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Comput. Surv.*, vol. 43, no. 2, pp. 11:1–11:29, Feb. 2011.

[16] K. Inkumsah and T. Xie, "Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 297–306.

[17] Java PathFinder, http://babelfish.arc.nasa.gov/trac/jpf/.

[18] Android developers guide, http://developer.android.com/guide/topics/ fundamentals.html.

[19] 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*, ser. ISSTA 2012. Minneapolis, MN, USA: ACM, 2012, pp. 100–110.

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

[21] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 11, no. 2, pp. 256–290, 2002.

[22] Dalvik - code and documentation from android's VM team, http://code.google.com/p/dalvik/.

[23] Android emulator | android developers, http://developer.android.com/tools/help/ emulator.html.

[24] Android debug bridge, http://developer.android.com/tools/help/adb.html.

[25] A. M. Memon, M. E. Pollack, and M. L. Soffa, "Automated test oracles for guis," in *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering: Twenty-first Century Applications*, ser. SIGSOFT '00/FSE-8. San Diego, California, USA: ACM, 2000, pp. 30–39.

[26] L. White and H. Almezen, "Generating test cases for gui responsibilities using complete interaction sequences," in *Software Reliability Engineering, 2000. ISSRE 2000. Proceedings. 11th International Symposium on*, 2000, pp. 110–121.

[27] P. Mehlitz, O. Tkachuk, and M. Ujma, "Jpf-awt: Model checking gui applications," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 584–587.

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

[29] A. Memon, I. Banerjee, and A. Nagarajan, "Gui ripping: Reverse engineering of graphical user interfaces for testing," in *Proceedings of the 10th Working Conference on Reverse Engineering*, ser. WCRE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 260–.

[30] Android testing framework, http://developer.android.com/guide/topics/testing/ index.html.

[31] Robotium, http://code.google.com/p/robotium/.

[32] Ui automator, http://developer.android.com/tools/testing-support-library/index.html.

[33] Monekyrunner, http://developer.android.com/tools/help monkeyrunnerconcepts.html.

[34] Espresso, https://google.github.io/android-testing-support-library/docs/espresso/index.html.

[35] Robolectric, http://pivotal.github.com/robolectric/.

[36] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet," in *To appear at the 30th International Conference on Automated Software Engineering*, ser. ASE '15, 2015.

[37] Android monkey, http://developer.android.com/guide/developing/tools/monkey.html.

[38] R. Sasnauskas and J. Regehr, "Intent fuzzer: crafting intents of death," in *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*. ACM, 2014, pp. 1–5.

[39] D. Amalfitano, A. Fasolino, P. Tramontana, B. Ta, and A. Memon, "Mobiguitar–a tool for automated model-based testing of mobile apps," 2014.

[40] 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.

[41] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '13. Indianapolis, Indiana, USA: ACM, 2013, pp. 641–660.

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

[43] Android monkey recorder, http://code.google.com/p/android-monkeyrunner-enhanced.

[44] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "Reran: Timing-and touch-sensitive record and replay for android," in *Software Engineering (ICSE), 2013 35th International Conference on.* IEEE, 2013, pp. 72–81.

[45] C. S. Jensen, M. R. Prasad, and A. Møller, "Automated testing with targeted event sequence generation," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ser. ISSTA 2013. Lugano, Switzerland: ACM, 2013, pp. 67–77.

[46] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. Cary, North Carolina: ACM, 2012, pp. 59:1–59:11.

[47] R. Mahmood, N. Mirzaei, and S. Malek, "Evodroid: Segmented evolutionary testing of android apps," in *Proceedings of the 2014 ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '14. Hong Kong, China: ACM, November 2014.

[48] H. Zhu, P. A. Hall, and J. H. May, "Software unit test coverage and adequacy," *Acm computing surveys (csur)*, vol. 29, no. 4, pp. 366–427, 1997.

[49] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in *2007 Future of Software Engineering.* IEEE Computer Society, 2007, pp. 85–103.

[50] M. Young, *Software testing and analysis: process, principles, and techniques.* John Wiley & Sons, 2008.

[51] L. C. Briand and Y. Labiche, "A uml-based approach to system testing," in *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, ser. &#171;UML&#187; '01. London, UK, UK: Springer-Verlag, 2001, pp. 194–208.

[52] A. A. Andrews, J. Offutt, and R. T. Alexander, "Testing web applications by modeling with fsms," *Software & Systems Modeling*, vol. 4, no. 3, pp. 326–345, 2005.

[53] A. Marchetto, P. Tonella, and F. Ricca, "State-based testing of ajax web applications," in *Software Testing, Verification, and Validation, 2008 1st International Conference on.* IEEE, 2008, pp. 121–130.

[54] X. Yuan and A. M. Memon, "Using gui run-time state as feedback to generate test cases," in *Software Engineering, 2007. ICSE 2007. 29th International Conference on.* IEEE, 2007, pp. 396–405.

[55] Y. G. Kim, H. S. Hong, D.-H. Bae, and S. D. Cha, "Test cases generation from uml state diagrams," in *Software, IEE Proceedings-*, vol. 146, no. 4. IET, 1999, pp. 187–192.

[56] P. Ammann and J. Offutt, *Introduction to software testing.* Cambridge University Press, 2008.

[57] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08. Budapest, Hungary: Springer-Verlag, 2008, pp. 337–340.

[58] B. Dutertre and L. De Moura, "The yices smt solver," vol. 2, no. 2, 2006.

[59] P. Godefroid, M. Levin, D. Molnar *et al.*, "Automated whitebox fuzz testing," in *Proceedings of the Network and Distributed System Security Symposium*, vol. 9, 2008.

[60] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. San Diego, California: USENIX Association, 2008, pp. 209–224.

[61] S. Khurshid and D. Marinov, "Testera: Specification-based testing of java programs using sat," *Automated Software Engg.*, vol. 11, no. 4, pp. 403–434, Oct. 2004.

[62] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. Chicago, IL, USA: ACM, 2005, pp. 213–223.

[63] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-13. Lisbon, Portugal: ACM, 2005, pp. 263–272.

[64] C. S. Păsăreanu, N. Rungta, and W. Visser, "Symbolic execution with mixed concrete-symbolic solving," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11. Toronto, Ontario, Canada: ACM, 2011, pp. 34–44.

[65] Symbolic PathFinder, http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc.

[66] K. Sen and G. Agha, "Cute and jcute: Concolic unit testing and explicit path model-checking tools," in *Computer Aided Verification*. Springer, 2006, pp. 419–423.

[67] K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun, "jfuzz: A concolic whitebox fuzzer for java." in *NASA Formal Methods*, 2009, pp. 121–125.

[68] K. Sen, D. Marinov, and G. Agha, *CUTE: a concolic unit testing engine for C*. ACM, 2005, vol. 30, no. 5.

[69] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[70] N. Tillmann and J. De Halleux, "Pex–white box test generation for. net," in *Tests and Proofs*. Springer, 2008, pp. 134–153.

[71] Y. Lei and K.-C. Tai, "In-parameter-order: A test generation strategy for pairwise testing," in *The 3rd IEEE International Symposium on High-Assurance Systems Engineering*, ser. HASE '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 254–261.

[72] P. J. Schroeder, P. Bolaki, and V. Gopu, "Comparing the fault detection effectiveness of n-way and random test suites," in *Empirical Software Engineering, 2004. ISESE'04. Proceedings. 2004 International Symposium on*. IEEE, 2004, pp. 49–59.

[73] P. Flores and Y. Cheon, "Pwisegen: Generating test cases for pairwise testing using genetic algorithms," in *Computer Science and Automation Engineering (CSAE), 2011 IEEE International Conference on*, vol. 2, June 2011, pp. 747–752.

[74] P. J. Schroeder, P. Bolaki, and V. Gopu, "Comparing the fault detection effectiveness of n-way and random test suites," in *Proceedings of the 2004 International Symposium on Empirical Software Engineering*, ser. ISESE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 49–59.

[75] R. C. Bryce and C. J. Colbourn, "One-test-at-a-time heuristic search for interaction test suites," in *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '07. London, England: ACM, 2007, pp. 1082–1089.

[76] G. Seroussi and N. H. Bshouty, "Vector sets for exhaustive testing of logic circuits," *Information Theory, IEEE Transactions on*, vol. 34, no. 3, pp. 513–522, 1988.

[77] X. Yuan, M. Cohen, and A. M. Memon, "Covering array sampling of input event sequences for automated gui testing," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07. Atlanta, Georgia, USA: ACM, 2007, pp. 405–408.

[78] C. H. P. Kim, D. S. Batory, and S. Khurshid, "Reducing combinatorics in testing product lines," in *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*, ser. AOSD '11. Porto de Galinhas, Brazil: ACM, 2011, pp. 57–68.

[79] J. Petke, S. Yoo, M. B. Cohen, and M. Harman, "Efficiency and early fault detection with lower and higher strength combinatorial interaction testing," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering.* ACM, 2013, pp. 26–36.

[80] A. Calvagna and A. Gargantini, "Ipo-s: incremental generation of combinatorial interaction test data based on symmetries of covering arrays," in *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on.* IEEE, 2009, pp. 10–18.

[81] R. Mandl, "Orthogonal latin squares: an application of experiment design to compiler testing," *Communications of the ACM*, vol. 28, no. 10, pp. 1054–1058, 1985.

[82] Number of available Android applications - AppBrain, http://www.appbrain.com/stats/number-of-android-apps.

[83] N. Mirzaei, H. Bagheri, R. Mahmood, and S. Malek, "Sig-droid: Automated system input generation for android applications," in *Proceedings of the 26th International Symposium on Software Reliability Engineering*, ser. ISSRE '15. Washington, DC: IEEE, November 2015.

[84] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood, "Testing android apps through symbolic execution," *SIGSOFT Softw. Eng. Notes*, vol. 37, no. 6, pp. 1–5, Nov. 2012.

[85] R. Valle é-Rai, P. Co, E. Gagnon, L. Hendren, and V. Lam, P.and Sundaresan, "Soot - a java bytecode optimization framework," in *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research, CASCON'99*, 1999.

[86] A. Bartel, J. Klein, Y. LeTraon, and M. Monperrus, "Dexpler:converting android dalvik bytecode to jimple for static analysis with soot," in *Proc. of SOAP*, 2012.

[87] S. Anand, "Techniques to facilitate symbolic execution of real-world programs," Ph.D., Georgia Institute of Technology, 2012.

[88] EMMA, http://emma.sourceforge.net/.

[89] F-droid, https://f-droid.org/.

[90] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic execution for software testing in practice: preliminary assessment," in *Proceedings of the 33rd International Conference on Software Engineering.* ACM, 2011.

[91] S. Park, B. Hossain, I. Hussain, C. Csallner, M. Grechanik, K. Taneja, C. Fu, and Q. Xie, "Carfast: achieving higher statement coverage faster," in *Proceedings of the ACM SIGSOFT 20th Int. Symp. on the Foundations of Software Engineering*. ACM, 2012.

[92] D. R. Slutz, "Massive stochastic testing of sql," in *VLDB*. Citeseer, 1998.

[93] N. Mirzaei, H. Bagheri, J. Garcia, A. Sadeghi, and S. Malek, "Reducing combinatorics in gui testing of android apps," in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, ser. ICSE '16. Austin, Texas: IEEE, May 2016.

[94] S. Rapps and E. Weyuker, "Selecting software test data using data flow information," *Software Engineering, IEEE Transactions on*, vol. SE-11, no. 4, pp. 367–375, April 1985.

[95] L. Deng, N. Mirzaei, P. Ammann, and J. Offutt, "Towards mutation analysis of android apps," in *To appear in Proceedings of the 2015 IEEE International Conference on Software Testing, Verification, and Validation Workshops*, ser. ICSTW '15. Graz, Austria: IEEE, 2015.

[96] L. Deng, J. Offutt, P. Ammann, and N. Mirzaei, "Mutation operators for testing android apps," *Submitted to Elsevier Journal of Information and Software Technology (IST)*, 2016.

[97] Trimdroid project, http://www.sdalab.com/projects/trimdroid.

[98] J. Jeon and J. S. Foster, "Troyd: Integration testing for android," Department of Computer Science, University of Maryland, College Park, Tech. Rep. CS-TR-5013, August 2012.

[99] T. Nelson, S. Saghafi, D. J. Dougherty, K. Fisler, and S. Krishnamurthi, "Aluminum: Principled scenario exploration through minimality," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. San Francisco, CA, USA: IEEE Press, 2013, pp. 232–241.

[100] D. Hamlet and R. Taylor, "Partition testing does not inspire confidence (program testing)," *IEEE Trans. Softw. Eng.*, vol. 16, no. 12, pp. 1402–1411, Dec. 1990.

[101] Phantomreference java api doc, https://docs.oracle.com/javase/7/docs/api/java/lang/ref/phantomreference.html.

[102] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev, "Static control-flow analysis of user-driven callbacks in Android applications," in *International Conference on Software Engineering*, 2015, pp. 89–99.

[103] R. Mahmood, "An evolutionary approach for system testing of android applications," 2015.

Curriculum Vitae

Nariman Mirzaei started his PhD with the Department of Computer Science at George Mason University (GMU) in 2008. His current research mainly focuses on software testing, program analysis, and mobile/distributed software systems. Nariman received his Bachelor's degree in Computer Science from Amirkabir University of Technology in Tehran, Iran in 2007 and respectively his M.S. Degree in Computer Science from Indiana University–Bloomington in 2009 and became a proud Hoosier.