AN EVOLUTIONARY APPROACH FOR SYSTEM TESTING OF ANDROID APPLICATIONS

by

Riyadh Mahmood

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

Committee:

_____	Dr. Sam Malek, Dissertation Director
_____	Dr. Songqing Chen, Committee Member
_____	Dr. Kris Gaj, Committee Member
_____	Dr. Angelos Stavrou, Committee Member
_____	Dr. Stephen G. Nash, Senior Associate Dean
_____	Dr. Kenneth S. Ball, Dean, Volgenau School of Engineering

Date: _____ Summer Semester 2015
George Mason University
Fairfax, VA

An Evolutionary Approach for System Testing of Android Applications

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

By

Riyadh Mahmood
Master of Science
Virginia Tech, 2004
Bachelor of Science
Virginia Tech, 2000

Director: Sam Malek, Associate Professor
Department of Computer Science

Summer Semester 2015
George Mason University
Fairfax, VA

Dedication

This is dedicated to my parents.

Acknowledgments

This dissertation would not have been possible without the endless support from my family.

I would like to thank my adviser Dr. Sam Malek for his kind support, guidance, patience and for not giving up on me. Sam's encouragement and direction gave me the confidence to go the last mile.

I thank Dr. Songqing Chen, Dr. Kris Gaj, and Dr. Angelos Stavrou for serving on my committee and providing me with insightful feedback. I really appreciate their flexibility in scheduling my exams.

I thank Nariman Mirzaei, Naeem Esfahani, and Thabet Kacem for their help. Finally, I thank my friends for their moral support and my colleagues for their understanding.

Table of Contents

List of Tables

List of Figures

Abstract

AN EVOLUTIONARY APPROACH FOR SYSTEM TESTING OF ANDROID APPLICATIONS

Riyadh Mahmood, PhD

George Mason University, 2015

Dissertation Director: Dr. Sam Malek

Mobile app markets have created a fundamental shift in the way software is delivered to the consumers. The benefits of this software supply model are plenty, including the ability to rapidly and effectively deploy, maintain, and enhance software used by the consumers. This paradigm, however, has given rise to a new set of concerns. Small organizations do not have the resources to sufficiently test their products, thereby defective apps are made available to the consumers of these markets. The situation is likely to exacerbate given that mobile apps are poised to become more complex and ubiquitous.

Automated testing of Android apps is impeded by the fact that they are built using an *application development framework (ADF)*. ADF allows the programmers to extend the base functionality of the platform using a well-defined API. ADF 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 ADF's sophisticated event delivery facilities. This hinders automated testing, as the app's control flow frequently interleaves with the ADF. At the same time, reliance on a common ADF provides a level of consistency in the implementation logic of apps that can be exploited for automating the test activities, as illustrated in this research.

Evolutionary testing technique has shown to be particularly effective for event driven software. In this research, I present the first evolutionary testing framework targeted at Android, called EvoDroid. Evolutionary testing is a form of search-based testing, where an individual corresponds to a test case, and a population comprised of many individuals is evolved according to certain heuristics to maximize the code coverage.

The most notable contribution of EvoDroid is its ability to overcome the common shortcoming of using evolutionary techniques for system testing. Evolutionary testing techniques are typically limited to local or unit level testing, as for system testing, they generate many invalid test cases. This occurs when the individuals (test cases) are crossed over to create new ones. The crossover does not consider which input and event genes are coupled to which part of the app, hence, it is not able to preserve the genetic makeup of parents in any meaningful way. I overcame this challenge by leveraging the knowledge of how Android specifies and constrains the way apps can be built. I devised a technique to analyze apps and infer models of their interface and behavior. Using these models my technique generates test cases reaching deep into the code in segments, i.e., sections of code that are amenable to evolutionary testing without the possibility of generating invalid test cases. Since a key concern in search-based testing is the execution time of the algorithm, I built EvoDroid to run the test cases in parallel on Android emulators deployed on the cloud, thus achieving several orders of magnitude improvement in execution time.

Chapter 1: Introduction

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

Platforms, such as Android, that have embraced this model of provisioning apps have seen an explosive growth in popularity. This paradigm, however, has given rise to a new set of concerns. Small organizations do not have the resources to sufficiently test their products, thereby defective apps are made available to the consumers of these markets. These defects are exploited with malicious intent compromising the integrity and availability of the apps and devices on which they are deployed. The situation is likely to exacerbate given that mobile apps are poised to become more complex and ubiquitous, as mobile computing is still in its infancy.

Automated testing of Android apps is impeded by the fact that they are built using an *application development framework (ADF)*. ADF allows the programmers to extend the base functionality of the platform using a well-defined API. ADF 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 ADF's sophisticated event delivery facilities. This hinders automated testing, as the app's control flow frequently interleaves with the ADF. At the same time, reliance on a common ADF provides a level of consistency in the implementation logic of apps that can be exploited for automating the test activities, as illustrated in this research.

The state-of-practice in automated system testing of Android apps is random testing. Android Monkey [10] is the industry's de facto standard that generates purely random tests. It provides a brute-force mechanism that usually achieves shallow code coverage. Several recent approaches [34,36,39,76,87,113],[42],[55],[69] have aimed to improve Android testing practices. Most notably and closely related to my work is Dynodroid [87], which employs certain heuristics to improve the number of inputs and events necessary to reach slightly better code coverage as that of Monkey.

Since prior research has not employed evolutionary testing and given that it has shown to be very effective for event driven software [71], [77], I set out to develop the first evolutionary testing framework targeted at Android, called *EvoDroid*. Evolutionary testing is a form of search-based testing, where an *individual* corresponds to a test case, and a *population* comprised of many individuals is evolved according to certain heuristics to maximize the code coverage. The most notable contribution of EvoDroid is its ability to overcome the common shortcoming of using evolutionary techniques for system testing. Evolutionary testing techniques [44, 77, 109, 110] are typically limited to local or unit testing, as for system testing, they are not able to promote the genetic makeup of good individuals during the search.

EvoDroid overcomes this challenge by leveraging the knowledge of how Android ADF specifies and constrains the way apps can be built. It uses this platform-specific knowledge to analyze the app and infer a model of its behavior. The model captures (1) the dependencies among the code snippets comprising the app, and (2) the entry points of the app (i.e., places in the code that the app receives external inputs). The inferred model allows the evolutionary search to determine how the individuals should be crossed over to pass on their genetic makeup to future generations. The search for test cases reaching deep into the code occurs in *segments*, i.e., sections of the code that can be searched independently. Since a key concern in search-based testing is the execution time of the algorithm, EvoDroid is built to run the tests in parallel on Android emulators deployed on the cloud, thus achieving several orders of magnitude improvement in execution time.

There are two models that support segmented evolutionary testing, the *Interface Model* (IM) and the *Behavior Model* (BM). The efficacy of EvoDroid depends on the quality of these models; and the quality of these models depend on the technique used to generate them. A particular technique may be fruitful in some instance, but not be suitable for other instances. For example, using static analysis to generate the models faces two issues, (1) it requires the source code to be available, and (2) using static analysis may generate incomplete models due to Android version fragmentation and variability of programming styles. Therefore, I provide an alternate models generation technique that extracts these models at runtime. The two alternate methods of models generation complement each other as each approach is susceptible to certain shortcomings. The static approach is largely a white box approach whereas the dynamic approach is mostly a black box approach.

I developed a prototype of EvoDroid that is able to operate in the static and dynamic configuration. I evaluated it by selecting a set of apps from the open source market F-Droid [19] and comparing the line coverage between EvoDroid, Monkey [10], and Dynodroid [87]. On average EvoDroid achieves 46% and 26% higher coverage than Monkey [10] and Dynodroid [87], respectively. EvoDroid static and dynamic approaches comparatively achieve similar code coverage depending on the quality of the models as fundamentally both algorithms apply a segmented evolutionary approach.

EvoDroid uncovered several defects during the experiments. For instance, EvoDroid found several cases of unhandled *number format exception* in Tipster, TippyTipper, and Bites that were due to either leaving the input fields empty, clicking a button that clears the input field followed by clicking a button that would operate on the inputs, or simply putting a string that could not be converted to a number. As another example, it found a defect in Bites, an app for finding and sharing food recipes, in which an unhandled *index out of bounds exception* would be raised when editing recipes without adding the ingredients list first.

Even though EvoDroid performs much better than existing tools, it does not achieve complete code coverage. Some of the reasons for not achieving complete coverage are

unsupported emulator functions, such as *camera*, as well as spawning asynchronous tasks that may fail, not finish by the time the test finishes, and thus not get included in the coverage results. Other reasons include code for handling external events, such as receiving a text message, dependence on other apps, such as calendars and contacts lists, and custom exception classes that are not encountered or thrown. Additionally, some of the applications contained dead code or test code that was not reachable, thus the generated static EvoDroid models would not be fully connected or the dynamic approach would not be able to reach a particular segment at run-time. Indeed, in many of these apps achieving 100% coverage is not possible, regardless of the technique.

It was not clear whether the observed accuracy and performance was due to the aforementioned issues, and thus an orthogonal concern, or due to the fundamental limitations in EvoDroid's approach to test generation. I, therefore, complemented the evaluation on real apps with a benchmark using synthetic apps. To ensure the synthetic apps are representative of the complexity of real apps, I identified properties pertinent to the complexity of Android apps and conducted an empirical study, by analyzing approximately 100 apps from an open source app market called F-Droid [19], and collected data on these property metrics. The data gathered from these complexity metrics are used by DroidGen, a synthetic app generator developed by me, to generate desirable test apps that exhibit one or more of these properties at varying degrees.

Using the synthetic apps, I benchmarked EvoDroid with Monkey [10] and Dynodroid [87] in terms of code coverage and execution time as the number of segments increase. I also compared EvoDroid against Monkey and Dynodroid in terms of complexity as it pertains to orchestrating sequences of events as well as input constraint satisfiability. EvoDroid outperforms both the Monkey and Dynodroid in terms of code coverage. For execution time, EvoDroid on a single instance runs slower. To mitigate this I developed a parallel testing framework where EvoDroid executes test cases on multiple Android emulators running simultaneously. In a parallel configuration, EvoDroid execution time is comparable to the Monkey, and much faster than Dynodroid.

I observed that the strength of EvoDroid is in reasoning about events. This is expected as evolutionary-based testing techniques excel in this arena. However, EvoDroid does not lend itself well when reasoning about input values and constraints. Especially when the constraint satisfiability drops to near zero. I explored an input generation solution, called SIG-Droid, that can be used to mitigate this weakness using symbolic execution. SIG-Droid combines program analysis techniques with symbolic execution [83] to systematically generate inputs for Android apps. To prune the domain of data inputs, SIG-Droid employs symbolic execution, a promising automated testing technique that can effectively deal with constraints. Symbolic execution uses symbolic values, rather than actual values, as program inputs. It gathers the constraints on input values along each path of a program execution, and with the help of a constraint solver generates actual inputs for all reachable paths.

This dissertation also presents possible applications of EvoDroid in terms of targeted testing. EvoDroid can be used to target specific areas of an app to obtain a predefined objective such as exploit generation, continuous integration test generation, as well as field failure replication. It is a mechanism to generate test cases to reach specific locations in code.

The main contributions of this dissertation are an automated Android testing framework, a method of applying evolutionary algorithms for generation of systems tests, a static analysis method for generating connected call graphs in event based systems, combining segmented evolutionary search and GUI crawling to generate dynamic models, and laying a foundation for subsequent research on targeted testing.

The rest of this dissertation is organized as following. I first provide background information on evolutionary algorithms and the Android platform in Chapter 2. I enumerate the related work in Chapter 3. In Chapter 4, I describe the problem and specify the scope of this thesis. Then I provide the overview of my approach in Chapter 5 and discuss the implementation choices of a static and a dynamic approach to segmented evolutionary testing. The details of the static approach are provided in Chapter 6 while the details of the dynamic approach are provided in Chapter 7. I describe the test execution environment on the cloud

and the cloud test manager application in Chapter 8. The details of a tool I developed on Android app evaluation subject generation are provided in Chapter 9. In Chapter 10, I describe the results of the evaluation of the static and dynamic segmented evolutionary testing as well as comparisons to other Android testing tools. Potential applications of this research are presented in Chapter 11. Chapter 12 provides details of an input generation tool for Android to aid in these applications. Finally, I conclude this dissertation with a summary of contributions and future work in Chapter 13.

Chapter 2: Background

In this chapter I provide an overview of evolutionary algorithms, the Android framework, as well as the Android emulator to help the reader follow the discussions that ensue.

2.1 Evolutionary Algorithms Background

Evolutionary algorithms are meta-heuristic optimization techniques that are inspired by biological evolution [43]. In my research, I use *genetic algorithm*, one of the most popular types of evolutionary algorithm. It is particularly shown to be useful for finding solutions in a large search space. Candidate solutions to the problem are called *individuals*. A solution is found by evolving *population* of individuals. Individuals are made up of *genes* representing the different elements of a solution.

As shown in Figure 2.1, evolutionary search entails three steps: (1) for each individual comprising the next generation of population, two individuals are selected from the current population with a likelihood that is proportional to their *fitness values*, indicating the quality of individuals. A *fitness function* is used to calculate the fitness value of individuals. This is the primary mechanism through which individuals with good genes are promoted in the evolutionary process. (2) *Crossover* between the two selected individual parents is performed to breed a new individual. A crossover is often realized by combining a subset of each parent's genetic makeup. The initial generation of individuals is made up of gene values selected at random, since no parents exist. (3) Finally, the new individual is *mutated*, meaning that a small part of its genetic makeup is modified in a random fashion. The evolutionary search is continued over generations until an end criterion is met (e.g., time limit, certain level of optimality, etc.).

2.2 Android Background

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) [16] for executing programs written in Java. Android also comes with an *application development framework* (ADF), which 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. Apps publish their capabilities and others can use them subject to certain security constraints.

Android apps are built using a mandatory XML *manifest* file. The manifest file values are bound to the application at compile time. This file provides essential information to an Android platform for managing the life cycle of an application. Examples of the kinds of information included in a manifest file are descriptions of the app's components among other architectural and configuration properties. Components can be one of the following types: *Activities*, *Services*, *Broadcast Receivers*, and *Content Providers*.

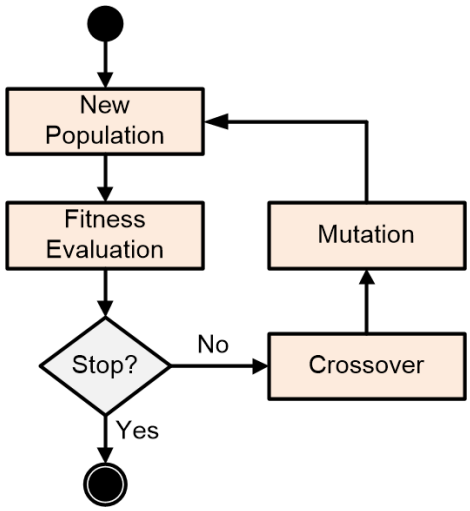


Figure 2.1: Evolutionary Process.

All major components, including Activity and Service, follow pre-specified lifecycles [6] managed by the ADF. For instance, Figure 2.2 shows the events in the lifecycle of an Activity: onCreate(), onStart(), onResume(), onPause(), onStop(), onRestart(), and onDestroy(). The lifecycle of the Service components is shown in Figure 2.3. The lifecycle event handlers are called by the ADF and play an important role in my research as explained

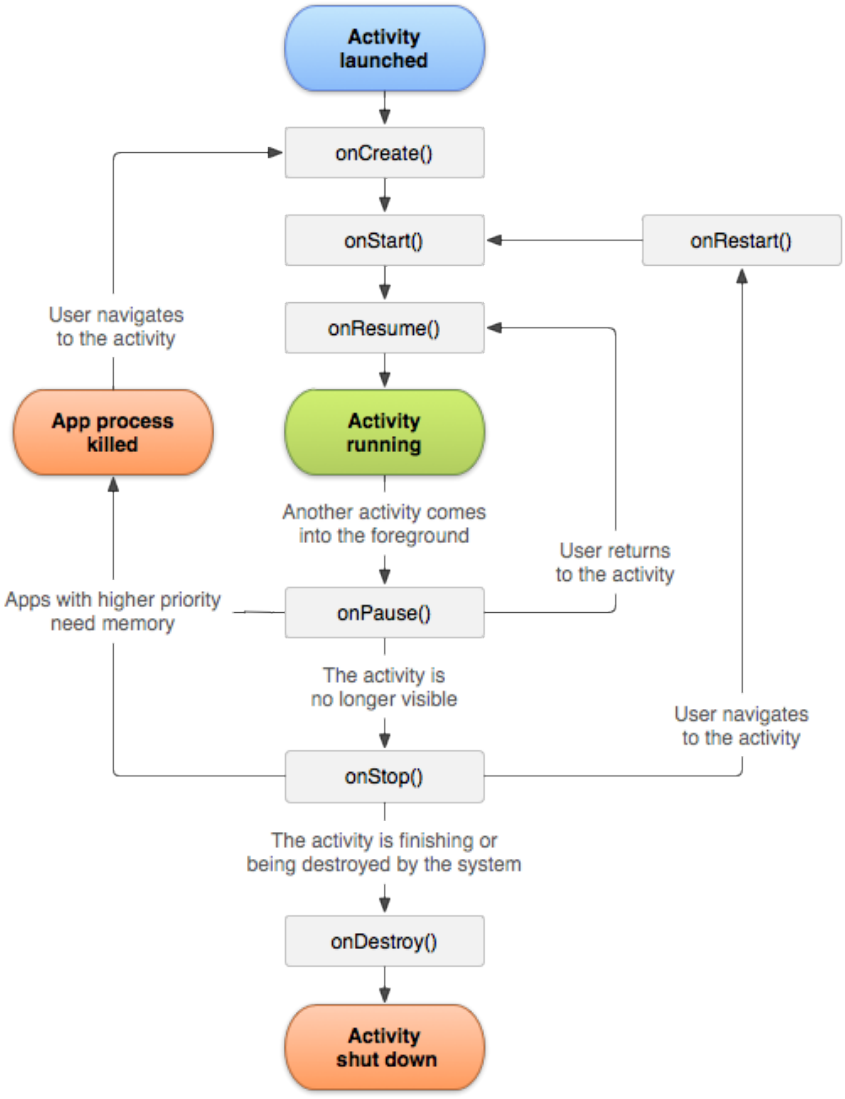


Figure 2.2: Android Activity Lifecycle [4].

later.

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 its controls are typically described in a configuration XML file with each layout and control having a unique identifier. A Service is a component that runs in the background and performs long running tasks, such as playing music. Unlike

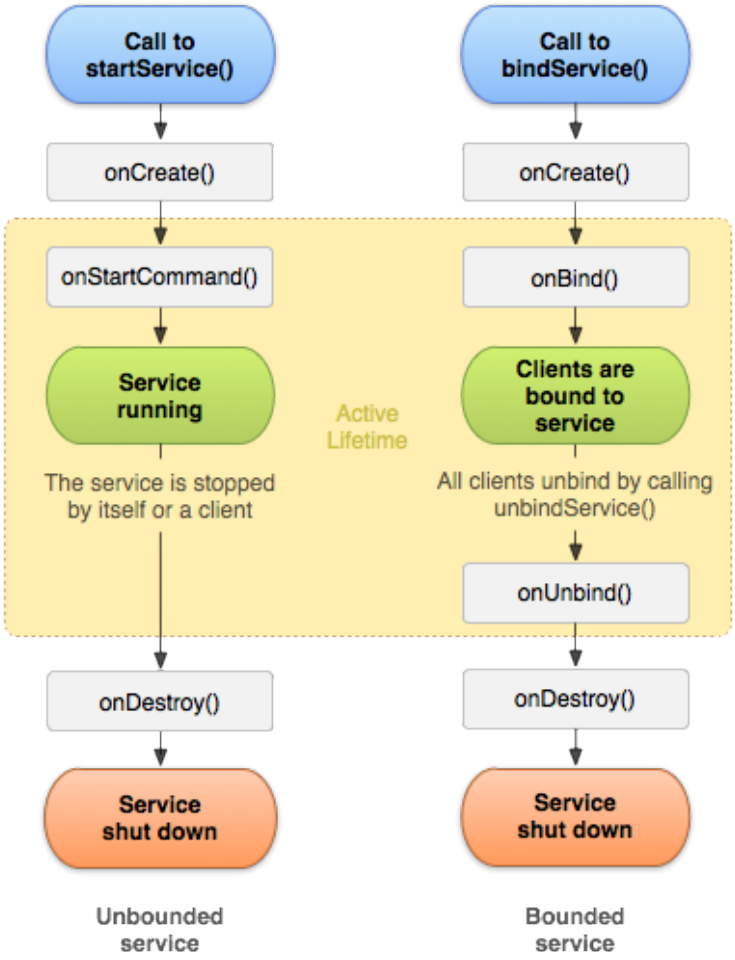


Figure 2.3: Android Service Lifecycle [12].

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. A Broadcast Receiver responds to system wide announcement messages, such as the screen has turned off or the 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 made possible through Android's messaging service.

2.3 Android Emulator

Emulators are programs that allow one to simulate and imitate devices when they are not available and when it is necessary to create different variations and models of the devices without having access to the physical devices.

The Android SDK includes an emulator [7] to imitate various mobile devices as well the different distributions of Android. As seen in the screenshot in Figure 2.4, it mimics a real Android device and allows the user to perform *most* of the possible tasks on a real device.

I have used the Android emulator 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) [5] tool is used to communicate with the emulators using a command line interface on the host machine.

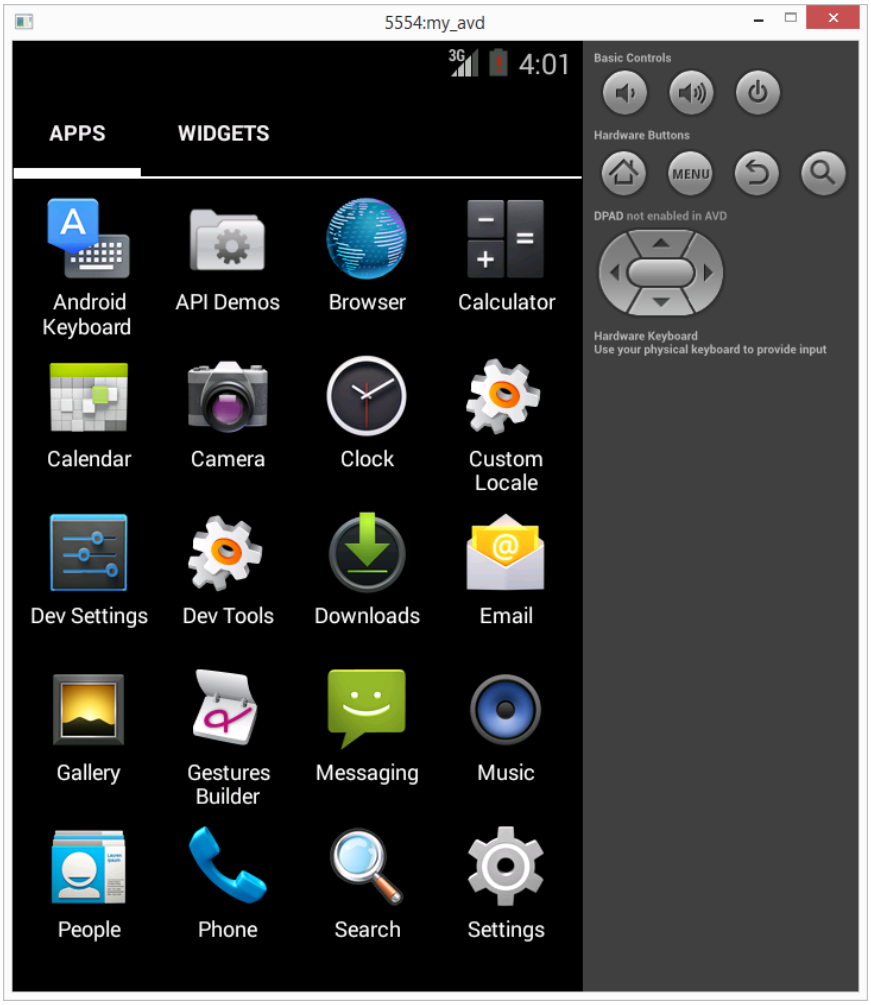


Figure 2.4: Android Emulator Screenshot.

Chapter 3: Related Work

Over the past decades, researchers and practitioners have developed a variety of methodologies, frameworks, and technologies intended to support the construction and the automation of software testing [37].

I provide an overview of the most notable research in this area and describe the seminal and closely related work.

3.1 Evolutionary Testing

Evolutionary testing falls under search based testing techniques and has typically been used to optimize test suites [71] [33] [102] [72] or has been applied at the class unit level [77]. Evolutionary testing approaches such as [71] [33] [102] [72] attempt to optimize a test suite for coverage. These are all blackbox approaches that build their GUI models at run time by executing and crawling or by recording user behavior, and therefore the generated models may not be complete. Since my approach uses program analysis and dynamically instruments the app, I obtain a more complete model of the apps behavior. They also differ from my research in that they represent test cases as genes. Two unit level evolutionary testing approaches are presented in [109] [110]. A combination of approaches are also presented in [77], and [44].

Evolutionary algorithm and symbolic execution techqniques are combined in [77], while evolutionary algorithm and hill climbing algorithm are used together in [44]. These techniques are all geared to unit testing. An ant colony optimization search is used in [45] along with a call graph similar to this research. However, their call graph is generated by executing the system and connecting overlapping call nodes to attempt to form the entire call graph. This suffers from the same issue as the GUI crawling methods, meaning that

the call graph model may be incomplete. Berndt et al. presents [46] a comparison of the performance of genetic algorithm-based test case generation.

3.2 Mobile Apps Testing

The Android development environment ships with a powerful testing framework [13] that is built on top of JUnit. Robolectric [28] is another framework that separates the test cases from the device or emulator and provides the ability to run them directly by referencing their library files. While these frameworks automate the execution of the tests, the test cases themselves still have to be written by the engineers. Many existing unit testing and system testing tools support automated test execution, but do not generate test cases automatically, such as MonkeyTalk [27], and Calabash [14].

Amalfitano et al. propose a crawling-based approach in [34] that leverages completely random inputs to generate unique test cases. Hu and Neamtiu in [76] present a random approach for generating GUI tests that uses the Android Monkey to execute. An automated Android GUI ripping approach is described in [36], where task lists are used to fire events on the interface to discover and exercise the GUI model. I use program analysis and instrumentation to derive the models. This sets this research apart from these works that employ manually created task lists for event generation. Moreover, these approaches have not considered the scalability implications of their solutions. Other similar GUI testing approaches are presented in [93] [73]. Automated black-box testing of smartphones are presented in [49] [116] and suffer similar setbacks as GUI testing. A grey-box model creation technique is presented in [113]; similar to my work, this approach is concerned with deriving models for testing of Android app and can potentially be substitued for my models. However, their model may be incomplete because they only focus on the GUI. A system level approach that combines symbolic execution with sequence generation is presented in [79], they attempt to find valid sequences and inputs to target locations. Their approach does not use a search based technique like mine, and they also work backwards from the targets. An approach

based on concolic testing of a particular Android library to identify the valid GUI events based on the pixel coordinates is presented in [38]. Dynodriod [87] is an input generation system for Android that is used extensively in my experiments.

Several prior approaches build on random testing techniques. Amalfitano et al. [36] described a crawling-based approach that leverages completely random inputs to generate unique test cases. MobiGUITAR [35] built on the top of the GUITAR [97] framework uses GUI ripping to build a model of an app. The model then is traversed by a depth-first search technique to generate test cases.

Other prior approaches have focused on the extraction of models for Android testing. *ORBIT* [113] is a grey-box model creation technique that creates a GUI model of the app for testing. A^3E [42] is a static taint analysis technique for building an app model for automated exploration of an app's Activities. SwiftHand [55] uses machine learning to learn a model of the app under test. There have also been efforts to extract graph models automatically by crawling the GUI [34, 93] and using those graphs to generate test sequences.

MonkeyRecorder [11] and RERAN [69] implement record and replay techniques for Android apps. MonkeyRecorder allows testers to record a script for GUI events, while RERAN logs the Android system commands events and turns them into runnable scripts. 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.

Jensen et al. [79] presented an approach that combines symbolic execution with sequence generation. The goal of their work is to find valid sequences and inputs to reach pre-specified target locations. Barad [64] symbolicaly executes a sequence of events to infer inputs for data components of the GUI. In my own research, I have developed techniques based on evolutionary search [88, 89], as well as symbolic execution [96] for testing Android apps.

Lin et al. [86] benchmark Dalvik Java code and Native code on real Android devices, whereas Kim et al. [82] present a storage benchmarking tool for Android based devices.

These approaches are not geared towards benchmarking of Android app testing tools. Li et al. [85] conducted an empirical study to discover quantitative and objective information about the energy behavior of apps that can be used by developers. To interact with an app, the authors utilize Monkey for test generation to estimate the energy cost of an app.

Combinatorial testing has shown to be an effective approach in GUI-based testing [114]. Approaches such as [61] propose using greedy or heuristic algorithms to generate minimal sets of tests for a given combinatorial criteria. Nguyen et al. [98] proposed an approach that leverages manually constructed behavioral models of app in pairwise testing of GUI-based applications. Kim et al. [81] introduced the idea of using static analysis to filter irrelevant features when testing software product lines.

A large body of research [90, 93, 94, 112, 115] has focused on testing traditional event-based systems. These approaches rely on techniques, such as ripping and crawling, to automatically extract directed graph models. Those models are then used to generate test sequences. However, data widgets are often abstracted away in such models. As such, system behaviors dependent on data values have not been adequately considered.

There has been a recent interest in using cloud to validate and verify software. TaaS is an automated testing framework that automates software testing as a service on the cloud [52]. Cloud9 provides a cloud-based symbolic execution engine [56]. Similarly, my framework is leveraging the computational power of cloud to scale evolutionary testing.

3.3 Android Testing Tools

This section provides further details on existing Android testing tools that are most closely related to my research. I provide an overview of these tools in the following subsections.

3.3.1 Android Monkey

Android Monkey [10] is a testing tool that generates random clicks, touches, or gestures, as well as a number of system-level events using the Android Debug Bridge [5] command line

tool. It is primarily used as a fuzz testing tool to stress test applications. The Monkey tool works with the Android emulator as well as real devices. It provides the options to specify the number of events to send the application, the type and frequency of those events, as well as any delay between events. The Monkey tool also also allows a set of debugging flags and specific package to target during testing.

The Monkey tool instruments and monitors the app under test as it fires off events. By default it stops on all unhandled exceptions and if the application under test stops responding. However, it is possible to configure the Monkey to continue sending events even after a crash. Although the state of the application may potentially be corrupt at this point. Additionally, it is possible to run the Monkey tool so it ignores security exceptions (e.g. permission errors).

This tool has wide adoption among academia and industry and is the de-facto standard in automated testing of Android apps in practice. It is simple to run and very fast.

3.3.2 MonkeyRunner

MonkeyRunner [26] is an unrelated tool from the Android Monkey that provides an API to control an emulator or device from outside of the Android code instead of relying on the ADB to shell into the emulator or device. With MonkeyRunner, it's possible to install an Android application or test package, run it, sends keystrokes to it, takes screenshots of its user interface, and stores screenshots on the workstation [26]. It is meant for functional and unit level testing, and the tests have to be written manually.

The MonkeyRunner tool is able to interact with multiple devices or emulators simultaneously and install apps, run user provided tests, and capture screenshots as results. It is suitable for running regression test suites and designed with extensible features. MonkeyRunner is uses Jython [23], an implementation of the Python programming language for Java, and provides three primary API abstractions: *MonkeyRunner*, *MonkeyDevice*, and *MonkeyImage*.

MonkeyRunner provides the functionality to connect to a device and and create user

interfaces. MonkeyDevice provides the functionality to install packages, start Activities, send events to an application, as well as run test packages. MonkeyImage provides the functionality to capture screenshots, convert images to various formats, as well as compare images as a form of comparing test results.

3.3.3 Dynodroid

Dynodroid [87] is an Android testing tool from researchers at Georgia Tech. It instruments the Android framework during testing and changes the distribution of events that are sent to the application under test. It uses less number of events than the Android Monkey tool but achieves higher code coverage. It uses three main components to test the applications: *Executor*, *Observer*, and *Selector*.

The executor component uses the ADB to send events to an emulator based on the event type. For GUI events it uses MonkeyRunner to send the events and for system events, it uses the Activity Manager tool [1] via Intent messages.

The observer component uses the Hierarchy Viewer tool [21] to instrument the GUI controls that are on the current screen. These serve as the GUI events. It also instruments the SDK to find what broadcast receiver and system services the app has registered for. These serve as the system events.

The selector component takes the information from the observer component and uses the executor component to execute the events. It uses deterministic as well as randomized events to test the application. For the random events it uses a uniform random and a biased random algorithm to choose events.

3.4 Mobile Apps Vulnerability Testing

There has been an increasing focus on vulnerability detection and testing. Static analysis techniques detect vulnerabilities by analyzing code or configuration without executing applications. Enck et al. [58] have developed a comprehensive catalogue of application-level security vulnerabilities in Android. ComDroid [54] statically analyzes inter-application

communication (IAC) to detect hijacking or injection vulnerabilities that are caused from incorrect use of IAC. SCanDroid [63] statically analyzes data flows to detect permission inconsistencies between apps that could possibly allow malicious access to sensitive information. AndroidLeak [65] statically analyzes information leak on Android. Other traditional open source or commercial static analysis tools such as FindBugs [20] and [75], Coverity [15], and Klockwork [24] could also be used to detect vulnerabilities in smartphone apps. While static analysis tools could be used to detect vulnerabilities during software development, they suffer from a high rate of false positives because of aliasing and context sensitivity [101] and [84].

Dynamic analysis techniques detect vulnerabilities at runtime. TaintDroid [57] detects information leak vulnerabilities using dynamic taint flow analysis at the system level. Kirin [59] and Saint [100] analyze configuration and runtime behavior of Android apps to enforce security policy and to allow only legitimate permissions. IPC Inspection [60] prevents privilege escalation at OS level. While static analysis can be performed on the whole code base, dynamic analysis can detect vulnerabilities that exist only on the executed path. Additionally, dynamic analysis can detect vulnerabilities only after software deployment.

In an effort to generate test inputs automatically, traditionally fuzz testing has been used. In fuzz testing, test input is generated either randomly or through modification of well-formed inputs. The idea of fuzz testing is to generate unexpected inputs to make the applications expose their vulnerabilities [108] [95] [99]. Naive fuzzing that randomly generates test input does not guarantee that the test input will penetrate all the execution paths, but variations of well-formed inputs could provide better code coverage [48] [62]. Fuzzing tools generate various types of input including file, user input, and network packets [107]. Monkey Runner [26] and IntentFuzzer [8] are GUI events fuzzers for Android. Still these are black-box approaches that do not utilize the code structure to generate test input. To further improve code coverage, white-box fuzzers generate test inputs based on information obtained from code analysis [103] [66] [68] [67] [51]. Most of these advanced approaches have been applied to only traditional desktop software, not smartphone applications. The

approach presented in [80] uses a black box approach that fuzzes Android GUI events as well as Hardware events by bypassing the Android runtime and accessing the underlying Linux kernel directly. Applying exhaustive approaches are typically not feasible due to the path explosion problem. AEG [41] generates test exploits using a combination of static and dynamic analysis techniques, but does so only for buffer overflow vulnerability. In addition, AEG is targeted for C programs, and not applicable to either Java or Android.

A fundamental difference between these fuzz testing approaches and this research is that while the traditional fuzz testing focuses random input and sequence generation to gain code coverage, my approach uses a systematic approach to generate tests to gain deep code coverage based on static analyses and dynamic instrumentation.

While above approaches detect vulnerabilities or prevent security attacks by analyzing and monitoring victim apps, a large body of studies [53] [50] [78] [105] [47] [111] [70] have been conducted to detect malware, including viruses and spywares, based on the observation of malicious behavior or code patterns. Shabtai et al. [104] extensively investigate existing mobile security approaches and suggest applicable approaches for Android, including intrusion detection, access control, and permission management techniques. SmartSiren [53] and Crowdroid [50] detect viruses by collectively analyzing communication activity information that was sent by agents in mobile devices to the central server for Windows Phone and Android, respectively. Andersson [40] presents a architecture for security testing.

Chapter 4: Research Problem

In this chapter, I present the problem that is the focus of this research. In the following sections, after describing the problem statement, I enumerate my research hypotheses.

4.1 Motivating Example

I use a simple Android app, called Expense Reporting System (ERS), to illustrate my research. The ERS app allows users to submit expense report from their Android devices. As shown in Figure 4.1, ERS provides two use cases that allow the user to create two types of report: *quick report* and *itemized report*.

When *quick report* is chosen, the user enters the expense item name and the amount, and subsequently presented with the summary screen. The user can choose to submit or quit the application on the summary screen.

The *itemized report* option presents the user with the option to enter the number of line items by tapping the plus and minus buttons. When *next* is tapped, the application prompts the user to enter the expense name and amount. This screen is repeated until all line items have been entered. Once all items are entered, the user is presented with a summary screen with the line items, their amount, and the total amount. The user can again choose to submit or quit the application at this time.

An interesting observation about the ERS is the *Line Item Count* screen. Here, we see that the *plus* button must be clicked more than the *minus* button followed by the *Next* button. This is an example of sequence of events that are necessary for automated search techniques to increase code coverage. While this is internal to a single screen, these types of event sequences are increasingly common when transitioning between screens. For example, there is a sequence involved when clicking the *Quick Report* then entering the *Expense Name*

and *Expense Amount* followed by the *Next* button. Without orchestrating these sequences properly, the automated technique will not be able to gain broad code coverage.

4.2 Problem Statement

Achieving high code coverage in GUI apps, such as the ERS, requires trying out a large number of sequences of events such as user interactions. My research is inspired by prior work [77] that has shown evolutionary testing to be effective when sequences of method invocation are important for obtaining high code coverage. However, application of evolutionary testing has been mostly limited to the unit level [44, 77, 109, 110], as when applied at the system level, it cannot effectively promote the genetic makeup of good individuals in the search.

Figure 4.2 illustrates the shortcoming of applying an evolutionary approach for system

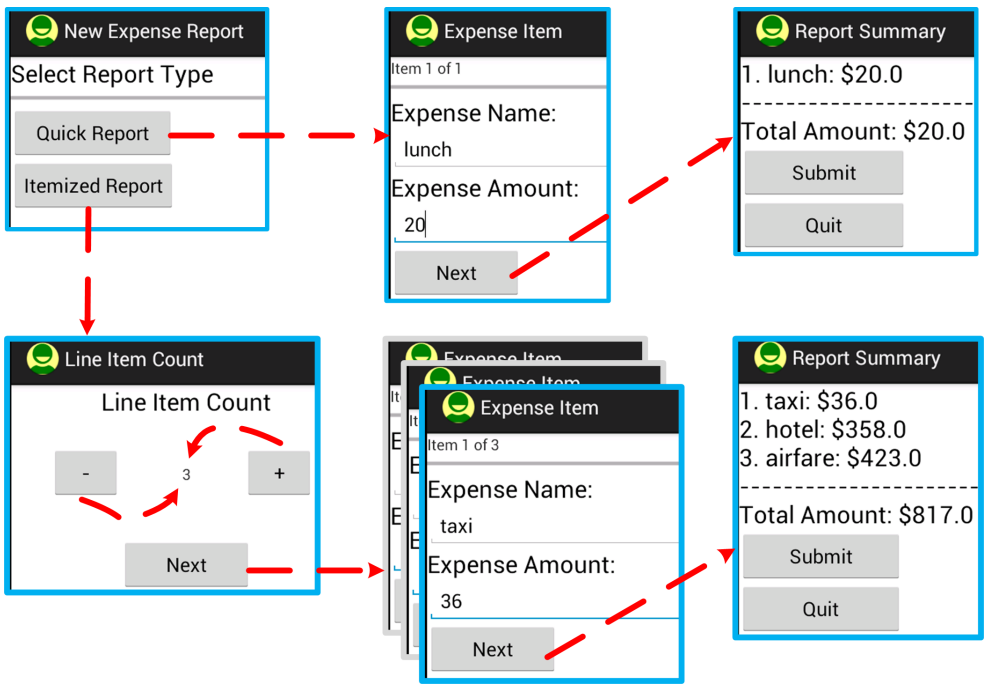


Figure 4.1: Expense Report System (ERS).

testing of a GUI application. In Figure 4.2b, we have two *individuals* in iteration 1 of the search. In this representation, an individual is comprised of two types of genes: *input genes* (e.g., values entered in text fields) and *event genes* (e.g., clicked buttons). The test case

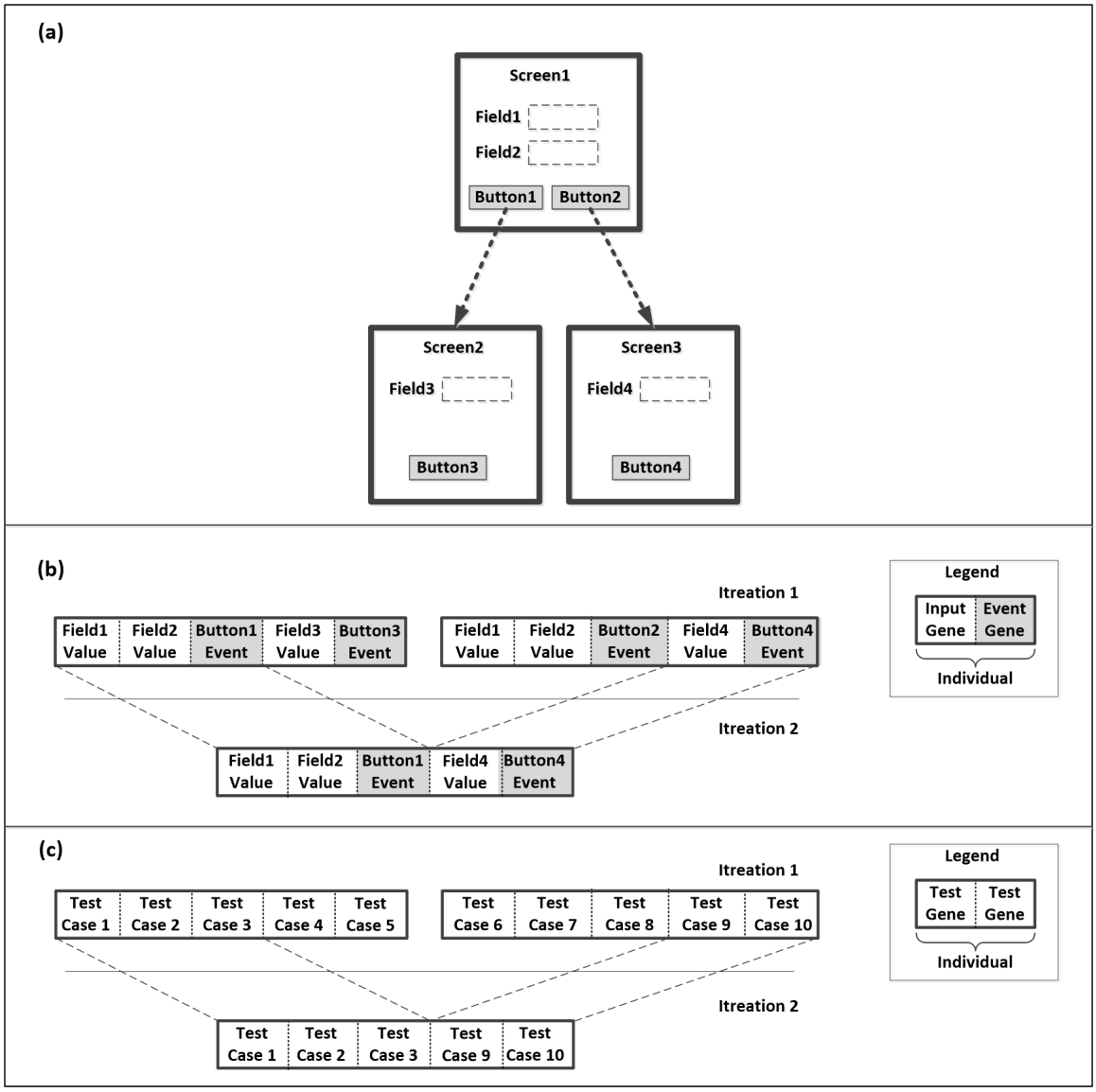


Figure 4.2: (a) An example app with three GUI components (b) a representation where the individual represents a test case, and (b) a representation where the individual represents a test suite

specified in an individual is executed from the left most gene to the right most gene. In essence, each individual is a test script.

Using the screens in Figure 4.2a, we can see that the two individuals in iteration 1 of Figure 4.2b represent reasonable tests, as each covers a different part of the app. For system testing, we would need to build on these tests to reach deeper into the code. The problem with this representation, however, is that there is no effective approach to pass on the genetic make up of these individuals to the next generations. For instance, from Figure 4.2b, we can see that the result of a crossover between the two individuals in iteration 1 is a new individual in iteration 2 that does not preserve the genetic makeup of either parents in any meaningful way. In fact, using the screens in Figure 4.2a, we can see that the tests cannot even be executed. There are two issues that contribute to this: (1) The crossover strategy does not consider which input and event genes are coupled to one another. For instance, the genes *Field1*, *Field2*, and *Button1* are all coupled with one another, as only together they can exercise the *Screen1* screen. Mixing genes from the other screens would be invalid to execute on *Screen1*. (2) The crossover strategy mixes genes from two different execution paths in the system, e.g. after clicking *Button1* on *Screen1*, the *Field4* and *Button4* are invalid as they are part of another execution path. Thus, it produces a test that is likely to be either not executable or inferior to both its parents. In evolutionary search, the inability to promote and pass on the genetic makeup of good individuals to the next generations is highly detrimental to its effectiveness.

To overcome the issues with this representation, prior approaches [33, 71, 72, 102] use evolutionary algorithm in conjunction with GUI exercising techniques. One such approach, called EXSYST [71], represents *test suites* as individuals and *tests* as genes, as depicted in Figure 4.2c. This approach generates tests that correspond to random walks on the GUI model. An individual is comprised of many random tests, i.e., each gene of the individual corresponds to a system test. EXSYST evolves the suites of tests to minimize the number of tests and maximize coverage. However, the probability of a single gene (test) achieving deep coverage remains the same as in the case of random testing. The overall coverage is no

better than the *initial population* (randomly generated tests), as the evolutionary algorithm is mainly used to minimize the number of tests.

4.3 Research Hypotheses

This research has investigated the following hypotheses [89]:

We can test the overall robustness of applications by exercising large portions of the application and maximizing code coverage. Evolutionary algorithms are sometimes used in automation of software testing. One of the issues faced by these approaches is representation of individuals. This is because it is difficult to capture all inputs and actions an application receives in a single individual. Performing cross-over on such an individual representation is problematic. Thus, to address this challenge, individuals are typically represented as test case suites, where each gene is a test case. This makes it difficult to automatically test entire applications; and it seems unachievable due to the unbounded inputs and large sequences of events that are necessary to gain coverage. I believe an approach to tackle this problem is to logically divide the app where external inputs and action events are supplied (e.g. for Android GUI activities these would be the *onCreate()* root nodes). I define each of these divisions to be a segment and evolve the test cases to maximize coverage for each segment independently. This overcomes the representation issue and allows us to maximize code coverage in Android apps.

> **Hypothesis 1:** *An evolutionary testing approach for system testing of Android apps can be devised.*

The efficacy of automated testing tools is often directly correlated of with the quality of the models that support them. There are two models that support segmented evolutionary testing: *Interface Model* (IM) and *Behavior Model* (BM). The quality of these models depend on the techniques used to generate them. A particular technique may be fruitful in some instance, but not suitable for other instances. For example, using static analysis to generate the models faces two issues, (1) it requires the source code to be available, and (2)

using static analysis may generate incomplete models due to Android version fragmentation and variability of programming styles. Therefore, alternate models generation techniques must be considered as static analysis may not always be applicable.

> **Hypothesis 2:** *Segmented evolutionary testing can be applied for system testing of Android apps regardless of the availability of source code.*

Evolutionary testing requires a large number of test cases to be executed. For segmented evolutionary testing, the number of test cases that need to be executed is multiplied by the number of segments as the evolutionary process is repeated in each segment. The search space for proper coverage is quite large in both of these instances. To add to this, Android test cases are executed within a supplied emulator that is known to be slow. Executing large number of test cases would seemingly be infeasible due to performance issues.

> **Hypothesis 3:** *Evolutionary testing for system testing of Android apps can be parallel processed to achieve scalability.*

Chapter 5: Approach Overview

Evolutionary testing is a form of search-based testing where an *individual* corresponds to a test case, and a *population* comprised of many individuals is evolved according to certain heuristics to maximize the code coverage. As described in Chapter 4, typically evolutionary testing is applied at the unit level [44, 77, 109, 110] as there is no effective approach to pass on the genetic make up of these individuals to the next generations at the system/app level.

To overcome the challenges with applying evolutionary testing at the system level, two aspects must be considered with respect to crossover. First, crossover cannot be allowed to span different components (e.g. GUI screens), and second, crossover cannot span different paths (i.e. sequence of GUI screen transitions). As shown in Figure 5.1, I identify sections of the app that can be searched independently (e.g. GUI screens), called segments, then the execution of the segments are arranged in the context of a path. An evolutionary algorithm is executed on each segment by using a model that aids us to form the representation of individuals for the segment as well as a model that aids us form the arrangement along the context of an execution path. Effectively, the process outlined in Figure 2.1 is repeated for each segment.[1] The goal of the algorithm is to maximize code coverage.

In Figure 5.1a, we see the first segment of the example app from Figure 4.2a. Here, an individual represents a test case and is formed by using the controls on the GUI screen. An evolutionary search is executed on this screen to generate test cases that maximize code coverage within the screen and potentially transitions to other screens or segments. Since the crossover is local to the screen we avoid mixing genes with other screens altogether. During the evolutionary search on this screen, a pool of ideal individuals are saved, shown as the α pool in Figure 5.1a. An ideal individual is defined to be a test case that, when executed, reaches the beginning of the next segment. Figure 5.1b shows the second segment where

[1]For the purposes of this dissertation I use the terms *screen*, *segment*, and *Activity* interchangeably.

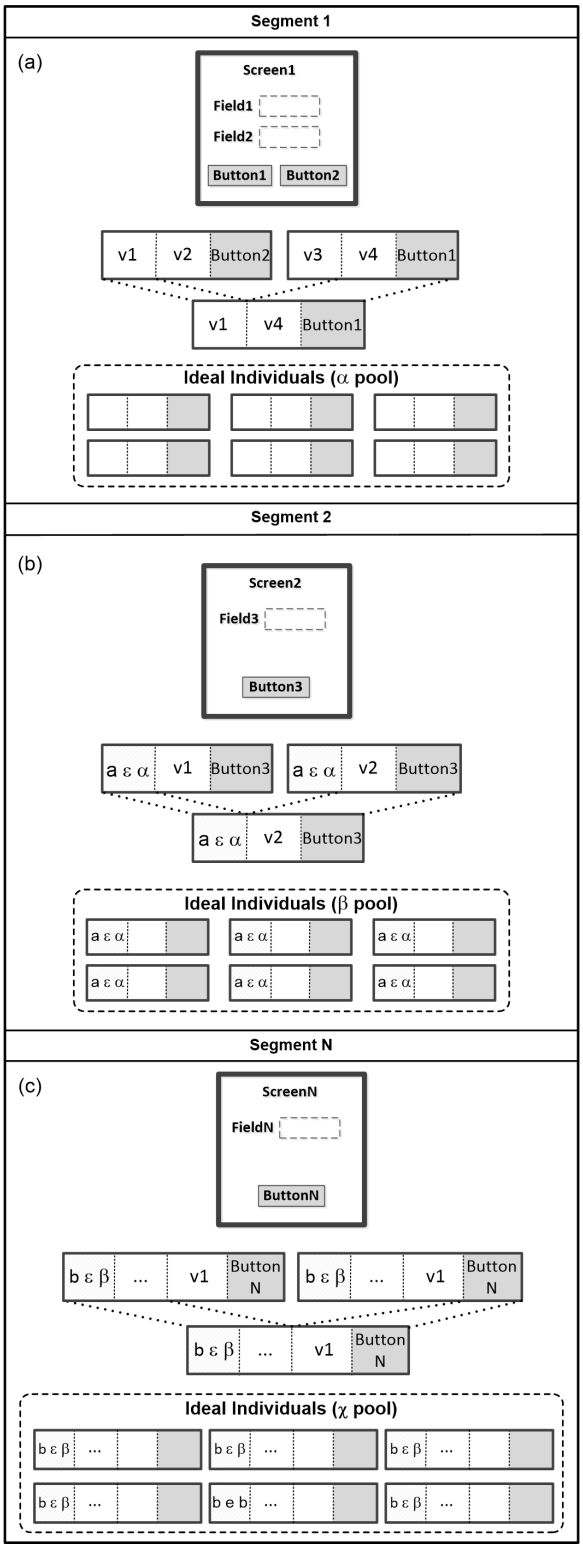


Figure 5.1: Segmented evolutionary testing: (a) segment 1 (b) segment 2, and (c) segment N. Each segment is searched independently with the previous segment used to reach the next.

another evolutionary search is executed in a manner similar to the first segment. However, an ideal individual from the α pool is pre-pended to the individuals in this segment. This ensures that we are able to reach the second segment. It should be noted that any of the ideal individuals from the α pool can be substituted to reach the second segment. Similar to the first segment, a pool of ideal of individuals are saved for the second segment, shown as the β pool in 5.1b. These ideal individuals combine genes from the first and second segment and reach the next segment when executed. The search continues in this manner deep into the app as shown in Figure 5.1c, essentially building up to a system level test case.

It should be noted that in segmented evolutionary testing, the test cases in each segment are fully executable test cases up to that point. In the case the search is abandoned before reaching the final segments, or if the final segments are not reachable, these test cases serve as the output of the algorithm.

The focus of this research is on generating test cases, not on whether the test cases have passed or failed. I acknowledge that automatically generating test oracles is a significant challenge that has been and continues to be the focus of many research efforts. However, I collect two types of results from the execution of tests that could help in the construction of oracles: any exceptions that may indicate certain software faults as well as code coverage information.

Even though this research is targeted towards testing GUI based applications such as Android apps, the techniques outlined here can be applied to any application that can be broken into segments and the necessary app models can be generated in each segment.

The remainder of this chapter describes the components used in the evolutionary process as well as the models that support them.

5.1 App Models

Segmented evolutionary testing requires two types of models to execute: Interface Model (IM), and Behavior Model (BM). The IM captures all of the inputs (e.g. GUI widgets) for each segment. It is used to form the representation of individuals in the evolutionary search.

The BM is used to guide the evolutionary search. It is used to identify and orchestrate segments in apps, measure fitness of individuals, and aid in reaching a particular segment. The specific type of BM depends largely on the implementation, and in this research two types of BM are described later.

5.2 Representation

As seen in Figure 2.1, the first step in the evolutionary process is to formulate the representation of individuals and their initialization. The models from Section 5.1 are used to determine the structure of genes for the individuals in each segment. The Interface Model (IM) tells us the inputs, their data type (such as integer, double etc.), the number of GUI elements (such as buttons), and/or system events relevant to the current segment.

An individual is represented as a vector shown in Figure 5.2a. Here, *previous segment* corresponds to the genes of an ideal individual from the previous segment, *Input [1..n]* corresponds to specific input values from the current segment, and *Event [1..m]* corresponds to the sequence of possible user actions or system events from the current segment. Each index in the vector contains a gene. The previous segment is a recursive relationship.

Both the number of input genes and the number of event genes are of variable length to handle the situations in which unexplored parts of the application require a certain sequence of events (e.g. a certain number of button clicks) to be executed.

The test case specified in an individual is executed from the left most gene to the right most gene including all previous segment genes, essentially traversing a path in the Behavior Model (BM).

5.3 Crossover

The first step in creating a new individual for the next generation is crossover. This process selects two individuals from the current population and creates a new individual by mixing their genetic makeup. I use a multi-point probabilistic crossover strategy. There is at least

one, and potentially multiple, crossover points between the two selected individuals.

The segment crossover probability is calculated as follows:

$$p(c) = \frac{1}{e^{(s-c)}} \tag{5.1}$$

where e is a configurable constant to achieve a decay factor, s is the index of the current segment being searched, and c is the index of a prior segment between 1 and s. The probability is *1.0* for the current segment, that is to say when $c = s$. This exponential decay function ensures that the earlier genetic makeup is not changed frequently, while

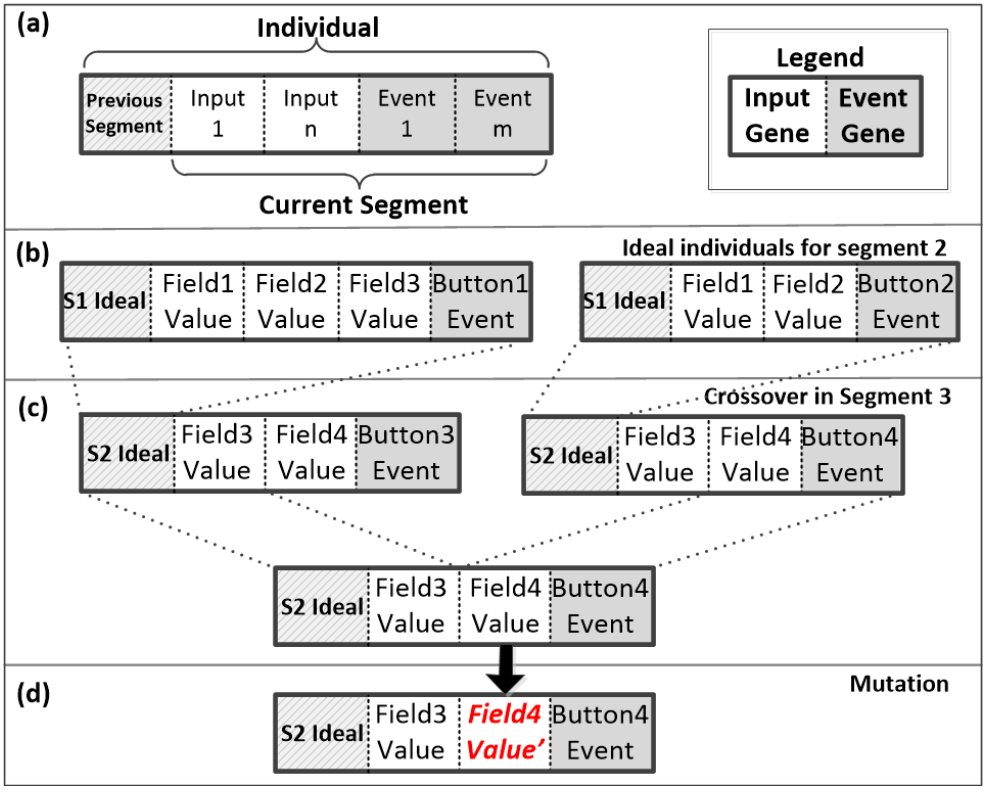


Figure 5.2: Segmented evolutionary testing: (a) representation of individual, (b) ideal individuals from segment 2, (c) crossover steps for creating an individual in the 3rd segment, and (d) mutation.

leaving the possibility open to find individuals that may explore new areas of the search space.

The crossover point for the current segment can be at any gene index and at most the length of the smaller of the two individuals. I only allow one crossover for the current segment, as this is sufficient to create variability in the new individual. Figure 5.2c shows the crossover steps for a pair of parent individuals in *segment 3* of an app. The newly created individual inherits part of the genetic makeup of the parents.

The previous segments are treated separately from the current segment and the probability function p(c) dictates the chance of crossover in each segment. There can potentially be a crossover at each of the previous segments, but I only allow swapping of the entire ideal individual for each segment, not in the middle of a previous ideal individual. Figures 5.2b and c show how ideal individuals found in prior segments are used to arrive at the parent individuals in Figure 5.2c. Here the new individual in Figure 5.2c inherits the previous segment individual from the left parent. If the probability function *p(c)* had dictated otherwise, it would have been inherited from the parent on the right.

This crossover strategy aims to preserve the genetic makeup of the solutions found for earlier segments, as I only allow the crossover to use the complete ideal individual for a given segment. Any ideal individual from that segment can be substituted, as they are all solutions for that segment. The previous segments for the new individual in Figure 5.2c share the same path (as the evolutionary process is applied within the context of a path in the BM), thus the structure of the individuals at each previous segment line up properly.

Note that this crossover strategy does not provide any guarantees that the input values and events satisfying an earlier segment in a path will be able to satisfy later segments in that path. For example, solving a particular constraint in segment *3* may require a specific value to have been entered in segment *1*. Indeed, the objective of the search is to find such combinations. The evolutionary search, guided by heuristics embedded in the fitness function, naturally weeds out sub-optimal tests. In addition, since I save many ideal individuals for each segment, each with different input/event genes, the algorithm is quite

effective at eventually discovering individuals that solve the entire path.

5.4 Mutation

Mutation changes parts of the genetic makeup of the newly created individual. Only the current segment genes are mutated with a probability threshold that is configurable. I mutate both input and event genes with several *creation*, *transformation* and *remove* operations.

The first type of mutation is done to the input genes of an individual. The creation of a numerical input includes boundary values, random, special/interesting values such as the number zero. For a string input, I generate purely random, uniformly distributed characters from the alphabet of a certain length, or *null*. Transformation operations for inputs include random value of same primitive data type, bit-flipping, arithmetic operations, and binary space reduction between boundary values. Removal operation for inputs is not applicable; they are included as *null* instead.

The second type of mutation is done to the event genes. The creation operator simply creates an event from the list of valid events specified in the IM. The number of added events is random with a minimum of one and a configurable upper threshold. Transformation operations for events include swapping event gene indexes, changing one event to another, and inserting a new event at a random index. Removal operation for events removes one or more event genes. The length of the overall individual can change as a result. Figure 5.2d show the mutation of a single gene to create a final unique individual that is different from both parents.

5.5 Fitness

A key aspect of evolutionary algorithms is the notion of fitness. In each generation, individuals are assessed for their fitness with respect to the search objective to be selected to pass on their genes. Fitness is assessed by applying a fitness function where the output value ranges from 0 to 1. It is a gradient from the least fit to the most fit.

In segmented evolutionary testing, the fitness is judged primarily by reaching the next segment, the uniqueness of the path traversed to reach the next segment compared to other individuals in the same generation, as well as the code coverage attained in the current segment. The exact fitness function formulation depends on the implementation as described later.

In both the static and dynamic implementations, a configurable number of test cases with the highest scores are directly copied to the future generations without any changes. This ensures the individuals with the best genetic makeup remain in the population.

5.6 Design Considerations

As stated earlier, I implemented EvoDroid to perform in two different modes of operation. In the first approach the models are generated via static analysis and in the second the models are discovered dynamically during runtime. Chapter 6 shows the implementation details of EvoDroid where the models are generated statically using program analysis, while Chapter 7 shows the implementation details of EvoDroid where the models are reflectively discovered dynamically at runtime. The two alternate methods of models generation complement each other as there are shortcoming to both approaches. The static approach is largely a white box approach, whereas the dynamic approach is mostly a black box approach.

It is not always feasible to generate static models due to various limitations. For example, using static analysis to generate the Interface Model (IM) relies on parsing XML files that define the GUI; but not all apps provide such XML files and instead define the GUI elements dynamically in code.. Similarly, the static Behavior Model used to identify the segments and run a segmented evolutionary algorithm along the context of paths is limited due to Android version fragmentation, variability of programming styles, third party libraries, code generation via reflection, and anonymous class declarations among other reasons. Often times the models generated statically are an over approximation and not all segments are reachable.

Dynamic generation of these models is subject to under approximation. Instead of

parsing XML files, this approach reflectively discovers the contents of the GUI. If parts of the GUI are not reached, a model for those parts would not be generated. Similar to the static approach, the dynamic method of discovering the model is also limited due to Android version fragmentation, variability of programming styles, third party libraries, code generation via reflection, and anonymous class declarations. The mechanism to reflectively identify GUI controls depends on the Android specifications. Furthermore, it may not always be feasible to instrument an application to extract the interface model as the controls may become active or inactive as a result of satisfying constraints. For example, a GUI screen may prompt the user to select a country from a drop-down menu. Based on the user's selection, the GUI may then activate an option for a zip code field if United States is selected in the first drop-down. Depending on the instrumentation mechanism, only the active controls may be extracted, missing the inactive controls (e.g. zip code) or alternatively extraneous controls (e.g. zip code) may be extracted.

Chapter 6: EvoDroid Static Approach

This chapter provides the implementation details of EvoDroid using static program analysis to generate models that support the segmented evolutionary testing process.

6.1 Approach Overview

The overall framework using statically extracted models is shown in Figure 6.1. The algorithm describing the process is shown in Algorithm 1. The input is an Android app's source code. From the source code, EvoDroid extracts two types of models, representing the app's external interfaces and internal behaviors, to automatically generate the tests: *Interface Model* (IM) and *Call Graph Model* (CGM). The CGM represents the Behavior Model in the static approach. The models are automatically extracted by analyzing the app's code.

IM provides a representation of the app's external interfaces and in particular ways in which an app can be exercised, e.g., the inputs and events available on various screens to

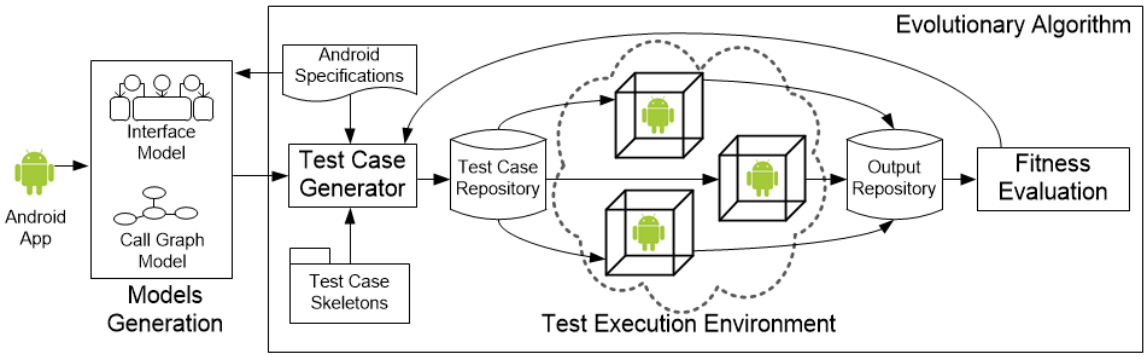


Figure 6.1: EvoDroid Static Approach Framework.

Algorithm 1: *static algorithm*

Input: $app \leftarrow AppUnderTest$
Output: $FinalTests \subset \mathcal{T}estCases$

```
1  FinalTests ← ∅;
2  IM ← ∅;
3  CGM ← ∅;
4  foreach screen ∈ app.screens do
5      foreach viewControl ∈ screen do
6          IM.add(screen, viewControl);
7  CGM ← app.methodCalls;
8  foreach eSourceNode ∈ CGM.nodes do
9      foreach eHandlerNode ∈ CGM.nodes do
10         if eHandlerNode.isFor(eSourceNode) then
11             CGM.connect(eSourceNode, eHandlerNode);
12 idealTests ← empty;
13 foreach path ∈ CGM.primePaths do
14     foreach screen ∈ path.screens do
15         foreach generation do
16             individual ← idealTests(prevScreen).append(IM.controls(screen));
17             population ← [individual0..individualN];
18             foreach ind ∈ population do
19                 testCase ← generateTest(ind);
20                 coverage ← executeTest(testCase);
21                 fitness ← evaluate(coverage);
22                 if (fitness == 1) then
23                     idealTests.add(screen, testCase);
24                 FinalTests.add(testCase);
25             if stoppingCriteriaMet then
26                 break;
```

generate tests that are valid for those screens. An example representation of an IM is shown in Figure 6.2b. EvoDroid uses the IM to determine the structure of individuals (tests), i.e., the input and event genes that are coupled together.

CGM is an extended representation of the app's call graph and represents the Behavior Model (BM) in the static approach. A typical call graph shows the explicit method call relationships. I augment that with information about the implicit call relationships caused by events (messages). An example of a CGM is shown in Figure 6.3. In a given application, all execution traces follow a certain path through the CGM. EvoDroid uses CGM to (1) determine the parts of the code that can be searched independently, i.e., *segments*, and (2) evaluate the fitness (quality) of different test cases, based on the paths they cover through the CGM, thus guiding the search.

Using these two models, EvoDroid employs a *segmented evolutionary testing* algorithm to automatically generate test cases. It aims to find test cases covering as many unique CGM paths from the starting node of an app to all its leaf nodes. In doing so, it logically breaks up each path into segments. It uses heuristics to search for a set of inputs and sequence of events to incrementally cover the segments. By carefully composing the test cases covering each segment into system test cases covering an entire path in the CGM, EvoDroid is able to promote the genetic makeup of good individuals in the search.

EvoDroid executes the automatically generated test cases in parallel, possibly on the cloud, to address scalability issues. The test cases are evaluated based on a fitness function that rewards code coverage and uniqueness of the covered path.

6.2 Apps Models Extraction

EvoDroid needs three types of information about the app under test for automatically generating test cases: (1) the genes comprising a valid individual, e.g., determining the input fields and GUI controls that should be paired up to have a valid test case for a GUI screen, (2) the app's segments, i.e., parts of the app that can be searched separately

to avoid the crossovers issues described earlier, and (3) the fitness value of different test cases. I developed Android-specific program analysis techniques to infer two models that can provide EvoDroid with this information. Algorithm 1 uses white-box static analysis to generate two necessary models: the Interface Model (IM) (lines 4–6 of Algorithm 1) and the Call Graph Model (CGM) (lines 7–11 of Algorithm 1).

```
<LinearLayout xmlns:android="http://schemas.
    android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  >
   ...
   <EditText
       android:id="@+id/fieldOneId"
       android:layout_width="match_parent"
       android:layout_height="wrap_content">
       <requestFocus />
   </EditText>
   <EditText
       android:id="@+id/fieldTwoId"
       android:layout_width="match_parent"
       android:layout_height="wrap_content">
       <requestFocus />
   </EditText>
   <Button
       android:id="@+id/buttonOneId"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:text="One" />
   ...
</LinearLayout>
```

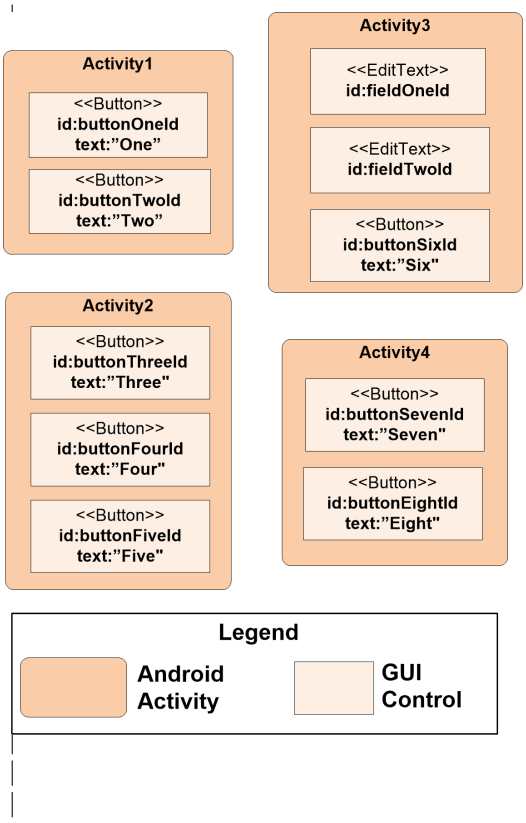


Figure 6.2: (a) Parts of the layout file for an Activity, (b) Example Interface Model.

6.2.1 Interface Model

The Interface Model (IM) provides information about all of the input interfaces of an app, such as the widgets and input fields belonging to an Activity. It also includes information about the application and system-level messages handled by each Activity. The IM is obtained by combining and correlating the information contained in the configuration files and meta-data included in Android APK (such as Android Manifest and layout XML files).

To generate the IM, I list all the Android Activities comprising an app with the help of information found in the Manifest file. Afterwards, for each Activity I parse the corresponding layout file. An example of such layout file is shown in Figure 6.2a. It is quite straightforward to obtain all information on each screen, such as widget type, name, and identifier from this XML document to generate the IM. Figure 6.2b depicts the IM for the Activities in Figure 6.2a. I use the information captured in IM to determine the structure (genes) of individuals for testing each component of the app.

6.2.2 Behavior Model

Android apps primarily communicate through the exchange of messages, called *Intents*, which contain information about actions to execute, associated data, and other meta-data of an event. Common types of Intents are *system events* (e.g., a device completes booting) and *GUI events* (e.g., indicating that an email should be sent). As mentioned, the Activity components have a pre-defined lifecycle, shown in Figure 2.2, corresponding to callbacks and events that a developer can leverage to enable different functionality at different points in the lifecycle. Each lifecycle callback acts as a different entry point for an Android app. As a result, each app has a set of disconnected call graphs, where each call graph represents a different context. Activities are activated via Intent messages. Intent messaging allows for late run-time binding between components, where the calls are not explicit in the code, rather made possible through Android's messaging service.

The Call Graph Model (CGM) is the Behavior Model in the static approach and contains a set of *connected* call graphs capturing the different possible invocation sequences within

a given application. I use MoDisco [25], an open source program analysis tool, to extract the app's call graph. However, since Android is an event driven environment, MoDisco generates *disconnected* call graphs for each app. Figure 6.3 shows an example CGM. As described later, I have extended MoDisco to infer the dashed lines to create a fully connected graph.

The root node of each call graph snippet is a method that no other part of the application explicitly invokes. There are two types of root nodes:

1. *Inter-component root nodes*: these root nodes represent methods in a component that handle events generated by other components or Android framework itself, e.g., an Activity generating a *StartActivity* event that results in another Activity's *onCreate()* method to be called, or the Android framework sending a *Resume* event that results in an Activity's *onResume()* method to be invoked.

2. *Intra-component root nodes*: these root nodes correspond to events that are internal to a component. For example, a Button on an Activity has a *Click* event associated with it. This event is handled by a class within the same Activity that implements the *OnClickListener* interface and overrides the *onClick()* method. These sorts of callback handlers are also root nodes, as they are called by the Android framework.

The inter-component root nodes are the logical break points for segments, and the inputs received at these nodes form the structure of individuals for the corresponding segments. We can determine the structure of this input using the IM. On the other hand, the intra-component root nodes do not mark a new segment, as they do not result in the execution to move to a different component (e.g., different Activity screen), and thus are not susceptible to the crossover problem.

Finally, for EvoDroid to generate tests and to determine their fitness, it needs the CGM to be fully connected. To that end, I have extended MoDisco with an Android-specific program analysis capability to infer the relationships among the disconnected nodes of the call graph. As depicted in Figure 6.3, I start with the *onCreate()* root node of the *main*

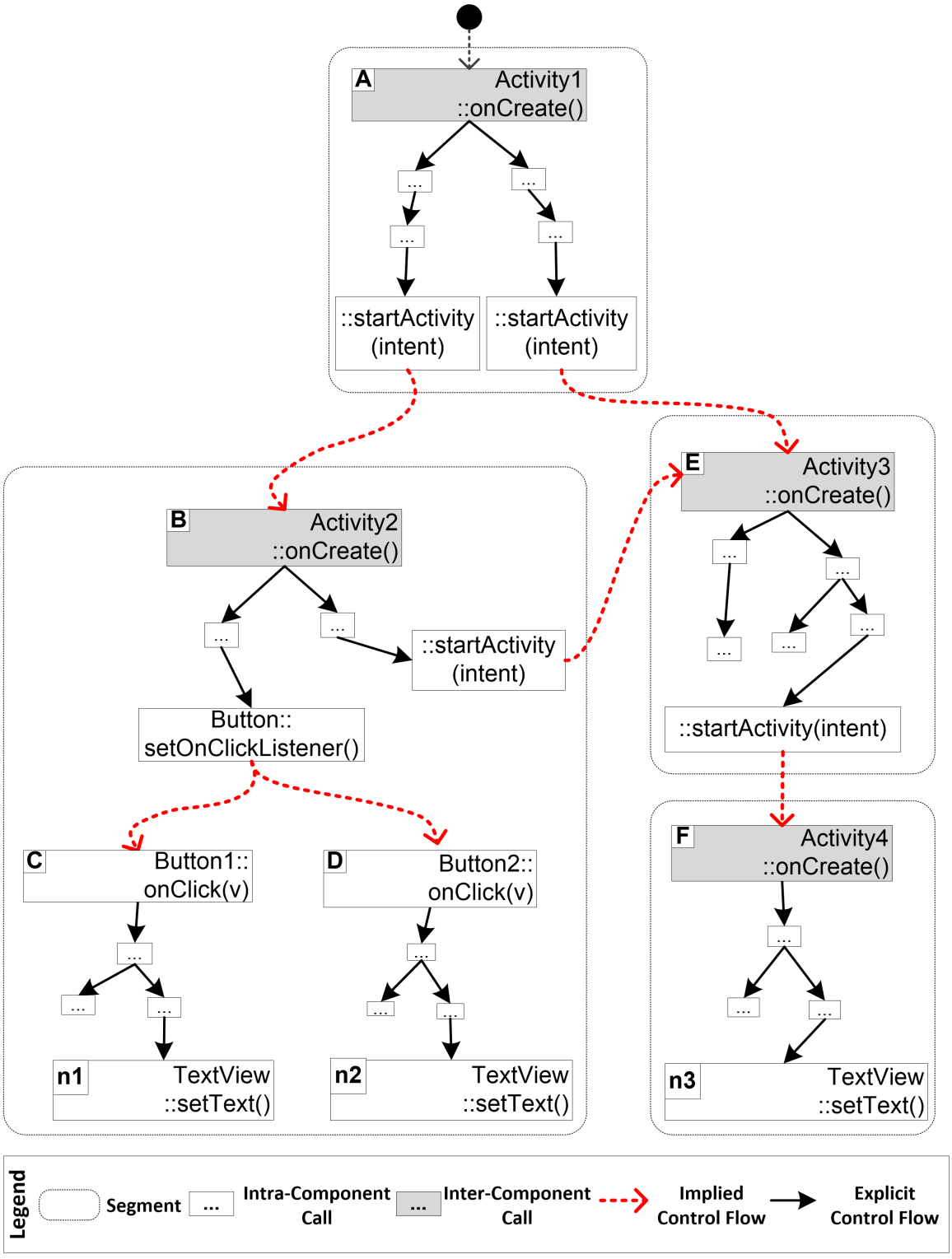


Figure 6.3: Example Call Graph Model.

Activity, which we know from Android's ADF specification to be the starting point of all apps. I then identify the Intent events and their recipients, as well as GUI controls and their event handlers, to link the different parts of the call graph and arrive at the final CGM. We know that the links would have to be to other root nodes, and achieved through sending of Intent events. The links formed as a result of this inference tell us the implied control flow as depicted with the dashed lines in Figure 6.3. In the case of inter-component events, the sender of Intent identifies its handler as one of the Intent's parameters. In the case of intra-component events, the root node responsible for handling that event is registered as a callback method with the sender.

As the call graph snippets are linked and connected, they are traversed in a similar fashion to arrive at the final connected CGM for the app.

6.3 Algorithm

The goal of EvoDroid is to find a set of test cases that maximize code coverage as shown in Algorithm 1. This is encoded as covering as many unique paths from the starting node of the CGM to its leaf nodes. For example, as depicted in Figure 6.3, it is to find test cases from node A to leaf nodes $n1$, $n2$, and $n3$. Two possible paths in this graph are $A \rightarrow B \rightarrow C \rightarrow n1$ and $A \rightarrow B \rightarrow E \rightarrow F \rightarrow n3$. The former involves two segments, while the latter involves four segments.

For each such path in Figure 6.3, EvoDroid starts from the beginning node and searches for test cases that can reach the leaf nodes (lines 12–26 of Algorithm 1). Each test case is represented as an individual in EvoDroid and its genes are the app inputs and the sequence of events. Unlike any prior approach, EvoDroid takes each path in the CGM, breaks it into segments, and runs the evolutionary search for each segment separately. Accordingly, the evolutionary process described here is repeated for each segment along each path in the CGM.

For each segment in each path, a population with a configurable number of individuals is generated. The evolutionary process is continued until all of the paths and their segments are

covered or a configurable threshold (e.g., time limit, certain level of code coverage, number of total test cases, etc.) is reached. The search is abandoned for a segment, and potentially a path, if the coverage is not improved after a configurable number of generations. This ensures the search does not waste resources on genes that cannot be further improved; it also prevents the search from getting stuck in infinite loops when there are cycles in the path. Fitness is measured based on how close an individual gets to reach the next segment as well the uniqueness of the covered path. With each iteration, EvoDroid breeds new individuals by crossing over current individuals selected with likelihood proportional to their fitness value, and then mutates them (e.g., changes some of the input values or events).

The *ideal* individuals from each segment are saved. An ideal individual is a test that covers the entire segment and reaches the root node of another segment. An ideal individual from the previous segment is prepended to the genes of a new individual for the next generation, as described in Figure 5.2. Essentially the test cases gradually build on the solutions found for the prior segments to build up to a system test case. A segment may also optionally be skipped if it was covered while attempting to cover another segment. For example, in Figure 6.3, since the segment $E \rightarrow F$ is shared in the following two paths $A \rightarrow B \rightarrow E \rightarrow F \rightarrow n3$ and $A \rightarrow E \rightarrow F \rightarrow n3$, it would only need to be evolved once (assuming ideal individuals were found the first time). Similarly, if while evolving $A \rightarrow B$, the algorithm inadvertently reaches $A \rightarrow E$, those ideal individuals are saved and EvoDroid may optionally skip solving $A \rightarrow E$.

The maximum number of individuals or test cases executed in the search process can be calculated as follows:

$$T = \sum_{i=1}^{|path|} Seg_i \times gen_{seg_i} \times pop_{gen_{seg_i}} \tag{6.1}$$

where $|path|$ is the number of unique paths (from the starting node to the leaf nodes) in CGM, seg is the number of segments for each path, gen is the number of generations per

segment, and *pop* is the population or the number of individuals per generation.

6.3.1 Fitness

EvoDroid considers two factors when assessing the fitness of individuals. The first is the distance traveled (number of nodes covered between segments) to reach the next segment, and the second is the uniqueness of the path covered compared to the other individuals in the same generation. The fitness of an individual i is determined as follows:

$$f(i) = \left(\frac{x}{n}\right) + u(i) \tag{6.2}$$

where x is the number of covered nodes in the path to the destination segment, n is the total number of nodes in the path to the destination segment, and $u(i)$ is the uniqueness function of the individual as follows:

$$u(i) = \left(1 - \left(\frac{x}{n}\right)\right) \times \sum_{k=1}^{l} \left(\frac{unique(r_k)}{l+k}\right) \tag{6.3}$$

where r_k is the covered node at index k in the path covered by the individual, and $unique(r_k)$ is 1 if the covered node at index k is unique compared to other individuals' coverage at the same index, and 0 otherwise, and l is the length of the path that this individual has covered.

When an individual for a given segment covers the entire segment path, I identify it as an *ideal* individual for that segment with a fitness score of 1. Of course, this means that there can be multiple individuals per generation that are ideal for a segment.

For illustration of how fitness is calculated, consider the hypothetical example depicted in Figure 6.4. It shows that the total distance (number of nodes) to reach *Activity1* from *Activity2* is 5. When the first individual executes, the shaded nodes are marked as covered by that individual. It covers 3 out of the 5 nodes along the path to segment root node, so it gets a distance score of $x/n = 3/5 = 0.6$, a uniqueness score of $u(i) = 0.4 \times 1/(4+1) + 1/(4+2) + 1/(4+3) + 1/(4+4) = 0.25$ as the entire path is unique at this time, for a total

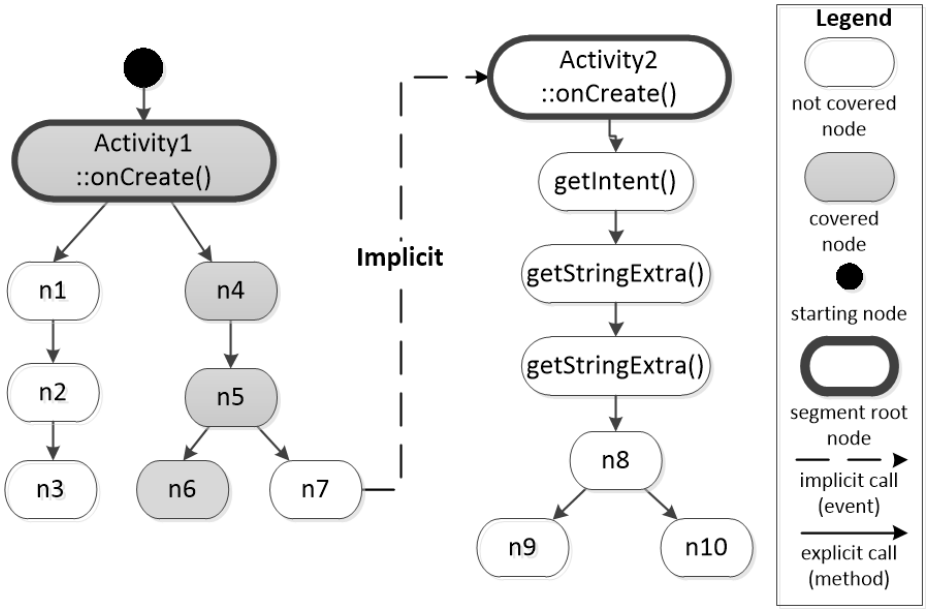


Figure 6.4: Fitness evaluation.

fitness score of $f(i) = 0.85$.

If another individual test case is executed but covers the path with nodes *Activity1* $\rightarrow n1 \rightarrow n2 \rightarrow n3$, it would get a distance score of $x/n = 1/5 = 0.2$, and additionally a uniqueness score. The length of the path this individual covered is 4 and all but the first node are unique (i.e., $n1$, $n2$, $n3$), so the uniqueness score is $u(i) = .8 \times (1/(4+2) + 1/(4+3) + 1/(4+4)) = 0.34$. The total fitness score for this individual would be $f(i) = 0.2 + 0.34 = 0.54$. Although the individual did not cover much of the path to the destination segment node, it is awarded a fractional score as it may discover a new area of uncovered code.

Note that the formulation of eq. 6.2 and 6.3 ensures that the uniqueness score alone never makes the value of fitness function to be 1 without reaching the destination. This prevents an individual to be labeled ideal without first reaching the destination.

Chapter 7: EvoDroid Dynamic Approach

This chapter provides the implementation details of EvoDroid using dynamic analysis to generate models that support the segmented evolutionary testing process.

7.1 Approach Overview

The approach overview for EvoDroid using dynamic models is shown in Figure 7.1. The algorithm describing the process is shown in Algorithm 2. The input is an Android app. EvoDroid dynamically builds two types of models from the app: *Interface Model* (IM) and *Transition Graph Model* (TGM). The TGM represents the Behavior Model in the dynamic approach. These models represent the app's GUI interfaces and GUI connectivity, and are used to automatically generate the tests in the evolutionary algorithm.

The Interface Model (IM) provides a representation of the app's GUI controls, e.g., the

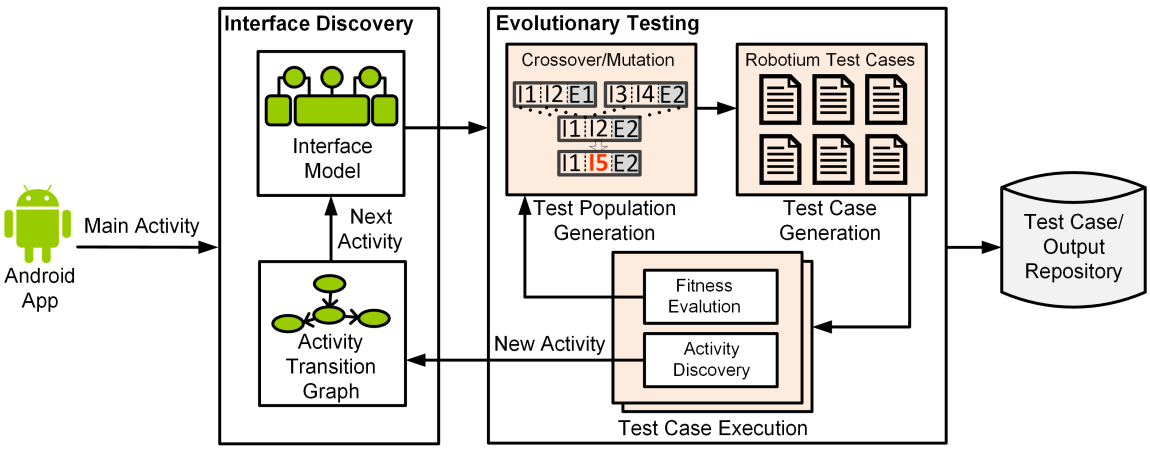


Figure 7.1: EvoDroid Dynamic Approach Framework.

Algorithm 2: *dynamic algorithm*

Input: $app \leftarrow AppUnderTest$
Output: $FinalTests \subset \mathcal{T}estCases$

```
1  FinalTests ← ∅;
2  IM ← ∅;
3  TGM ← app.firstScreen;
4  tranTests ← ∅;
5  foreach screen ∈ TGM.screens do
6      viewControls ← screen.getViewControls;
7      foreach viewControl ∈ viewControls do
8          IM.add(screen, viewControl);
9      foreach generation do
10         individual ← tranTests(screen).append(IM.controls(screen));
11         population ← [individual0..individualN];
12         foreach ind ∈ population do
13             testCase ← generateTest(ind);
14             coverage ← executeTest(testCase);
15             fitness ← evaluate(coverage);
16             if (fitness == 1) then
17                 idealTests.add(screen, testCase);
18             screenId ← currentScreen.getId;
19             if (screenId! = screen.screenId) then
20                 TGM.add(currentScreen);
21                 tranTests.add(currentScreen, testCase);
22             FinalTests.add(testCase);
23         if stoppingCriteriaMet then
24             break;
```

inputs and events available on various screens to generate tests that are valid for those screens. An XML representation of IM is shown in Figure 7.2b and the model is shown in 7.2c. EvoDroid uses the IM to determine the structure of individuals (tests), i.e., the input and event genes that are coupled together. The IM for each Activity is extracted dynamically by executing a special model generation test case on the Activity, called Discovery Test Case (DTC). The sole purpose of this test case is to extract the model.

The Transition Graph Model (TGM) provides a representation of how the app's GUI screens (i.e. Activities) are connected to one another. An example of a partial TGM is shown in Figure 7.3. All use cases in a given app follow a certain Activity transition path through the TGM. EvoDroid uses TGM to reach new Activities based on the paths the use cases traverse through the TGM.

Using these two models, EvoDroid employs a combination of evolutionary testing and GUI crawling technique. It aims to generate test cases that find and cover as many of the Activities as possible by dynamically crawling the application. In doing so, it logically executes an evolutionary algorithm on each Activity to discover other connected Activities.

EvoDroid starts with the first Activity in the app and adds it to the TGM. The TGM contains a set of nodes made up of Activities and edges that connect them. Initially it only contains the starting Activity, and subsequently, additional nodes are added as they are discovered. Then the DTC is used on the Activity to extract its IM. Using the IM, an evolutionary search is performed on the Activity in an attempt to maximize code coverage for that Activity. EvoDroid uses heuristics to search for a set of inputs and sequence of events to incrementally cover the Activity. In doing such a search, new Activities are discovered and added to the TGM. If complete coverage is attained for a given Activity, then we know for certain that all possible connected Activities have been discovered. The search continues in this manner until an evolutionary search has been run on all Activities in the TGM.

7.2 Apps Models Extraction

The dynamic version of EvoDroid needs two types of information about the app under test for automatically generating test cases: (1) the genes comprising a valid individual, e.g., determining the input fields and GUI controls that should be paired up to have a valid test case for an Activity, and (2) previous Activities that must be traversed to reach the current Activity. EvoDroid automatically generates the models that capture this information at run-time as described below. Thus, unlike the static version of EvoDroid that requires access to source code to extract a call graph model of the app, the dynamic approach is able to generate tests without requiring access to source code. Algorithm 2 discovers the Interface Model (IM) (lines 6–8 of Algorithm 2) and the Transition Graph Model (TGM) (lines 19–21 of Algorithm 2) reflectively at run time.

7.2.1 Interface Model

The Interface Model (IM) provides information about all of the input interfaces of an app, such as the widgets and input fields belonging to an Activity. The IM is obtained by executing a special test case (not part of the test cases that are executed in the evolutionary algorithm) which I call Discovery Test Case (DTC). This test case is executed once per Activity prior to running the evolutionary testing algorithm on it.

Figure 7.2a shows the IM extraction mechanism on Android using the DTC. The DTC is a Robotium [29] test case that reflectively identifies what is on the current screen. The test case logs the IM in an XML format in LogCat [9], the logging facility provided by Android. The XML representation of the IM is then extracted via the Android Debug Bridge (ADB). This mechanism is employed as EvoDroid runs outside of the Android emulator or device and does not have any permissions, visibility, or dependencies to read from the emulator or device storage.

The DTC recursively combs the current screen for GUI controls such as input boxes, buttons, radio buttons, image buttons, check boxes, lists, dialogs, context menu, action

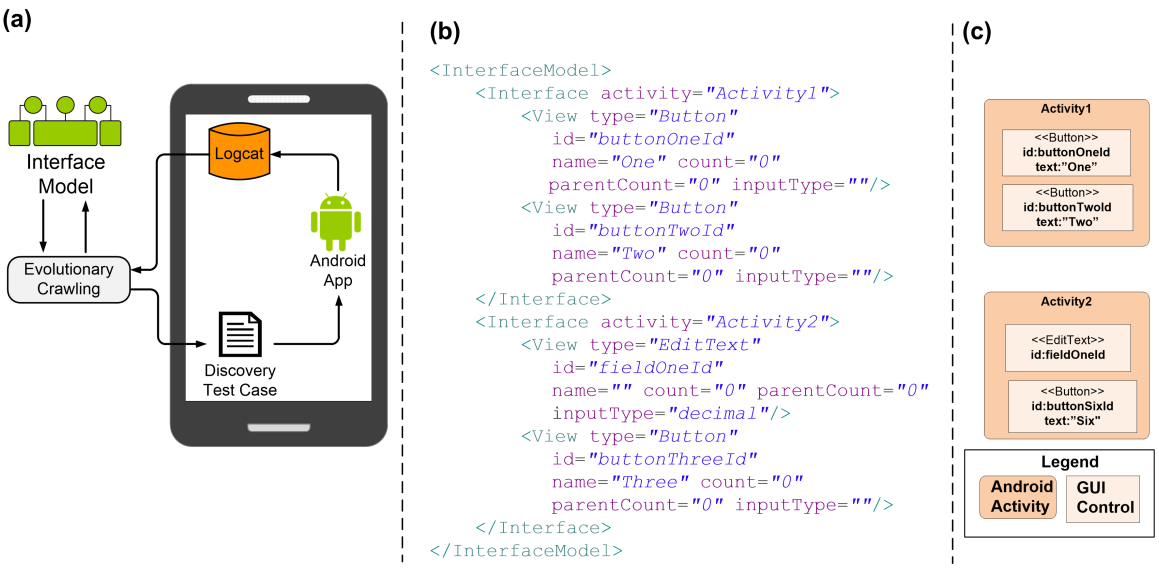


Figure 7.2: EvoDroid's (a) Interface Model extraction mechanism, (b) sample XML representation of a partial Interface Model, (c) sample partial Interface Model.

menu, physical button menu, tabs, seek bars etc. As these controls are encountered information about them are captured in an XML representation of the Interface Model. Some widgets are marked private by the core Android libraries, in such instances, I use reflection to access the desired information.

It is quite straightforward to obtain all information on each screen, such as widget type, name, and identifier to generate the IM. Figure 7.2b depicts an example IM in XML format as extracted by executing the DTC. This XML has a custom schema developed by me, and is *not* the same as the XML that is used to define Activity controls. I use the information captured in IM, shown in Figure 7.2c, to determine the structure (genes) of individuals for running the evolutionary search on each Activity.

7.2.2 Behavior Model

The Transition Graph Model (TGM) contains a set of Activities capturing the different possible GUI screen transitions within a given application. As mentioned before, TGM represents the Behavior Model in the dynamic approach. Figure 7.3 shows a partial TGM example.

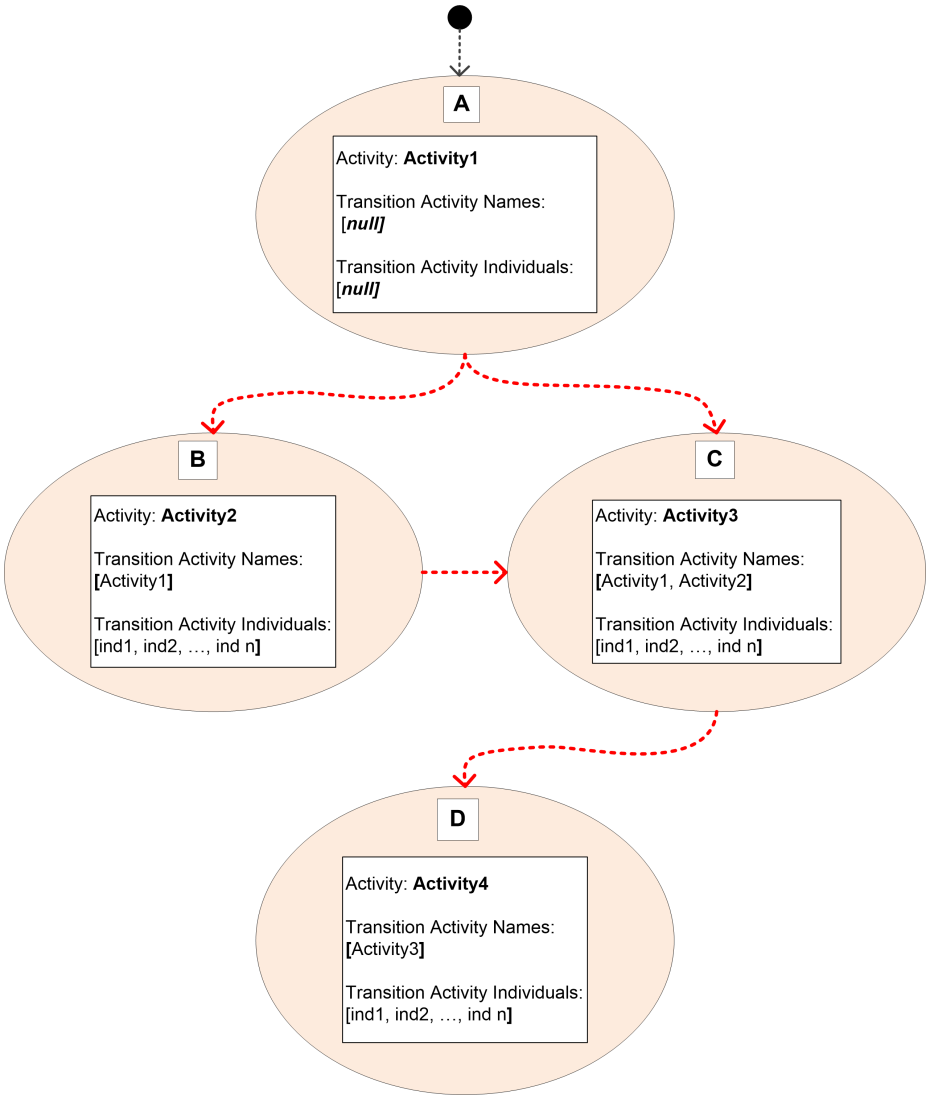


Figure 7.3: Partial Transition Graph Model example.

The TGM is constructed dynamically as new Activities are discovered during test execution. The evolutionary algorithm is executed on an Activity using its IM with the goal to attain full coverage of the Activity. During this process EvoDroid discovers new Activities as the transitions are made. This is different from Chapter 6 where I statically generate the transition paths and the transitions are known prior to applying the evolutionary search.

The TGM contains a set of connected Activity nodes. Each node is made up of the name of the Activity, the names of the previous transition Activities, as well as the inputs and sequence of actions and events (the path) that were executed to reach and transition to the current Activity. A node also has a flag to identify whether the evolutionary algorithm has been executed on it.

7.3 Algorithm

The goal of EvoDroid is to find a set of test cases that maximize code coverage (lines 9–24 of Algorithm 2). This is encoded as discovering and gaining code coverage of as many Activities as possible starting from the *main* Activity. In the context of the example depicted in Figure 7.3, it is to start crawling from node A and find test cases to cover nodes B, C, and D. These are the Activities that need to be crawled and covered.

For each Activity in Figure 7.3, EvoDroid executes and searches for test cases that cover as much of the Activity as possible. Each test case is represented as an individual in EvoDroid and its genes are the app inputs and the sequence of events (from the IM). Unlike any prior crawling approach, EvoDroid takes each Activity, extracts its Interface Model (IM) reflectively, and runs an evolutionary search. Accordingly, the evolutionary process described here is repeated for each Activity in the TGM.

A population with a configurable number of individuals is generated per Activity. The evolutionary process is continued until the all the known Activities are covered or a configurable threshold (e.g., time limit, certain level of code coverage, number of total test cases, etc.) is reached. The search is abandoned for an Activity, if the coverage is not improved

after a configurable number of generations. This ensures the search does not waste resources on genes that cannot be further improved. Fitness is measured based on how much of the Activity has been covered (number of lines has been covered). With each iteration, EvoDroid breeds new individuals by crossing over current individuals selected with likelihood proportional to their fitness value, and then mutates them (e.g., changes some of the input values or events).

An individual that discovers a subsequent Activity is saved as part of that new Activity as the transition Activity. When the evolutionary process is applied to the newly discovered Activity, this individual will be used to reach the new Activity. An individual can discover a new Activity without covering the entire Activity it is being executed on; in such cases only part of the individual's genes will be used to reach the newly discovered Activity.

Since individuals attempt to cover the entire Activity, discovering transitions to other Activities are implicitly encoded in the search. That is to say, if an individual is able to cover the entire Activity, then it has discovered all possible transition from the Activity. This is the primary crawling mechanism in EvoDroid. As these transitions occur, EvoDroid detects the new Activity and adds it to the Transition Graph Model (TGM) as a new Activity node. The new node contains the names of the transition Activities (including the Activity that is currently being searched and made the transition), the name of the new Activity, and the *individual* (inputs, events/actions genes) hat was executed to reach this new Activity.

Transition Activity genes are prepended to the genes of a new individual for the next Activity (as a transition genes), as described in Figure 5.2. Essentially the test cases gradually build on the solutions found for the prior Activities to crawl to new areas in the app. An Activity may optionally be skipped if it was discovered and covered through another execution path. For example, in Figure 7.3, since Activity C is shared in the following two Activity transition paths $A \rightarrow B \rightarrow C \rightarrow D$ and $A \rightarrow C \rightarrow D$, it would only need to be evolved once (assuming the entire Activity was covered the first time).

7.3.1 Fitness

The dynamic version of EvoDroid uses the number of source lines of code covered for the Activity when assessing fitness of individuals for the next generation. The fitness of an individual i is determined as follows:

$$f(i) = \left(\frac{x}{n}\right) \tag{7.1}$$

where x is the number of covered lines covered in the Activity, n is the total number of lines in the Activity.

It should be noted that the individuals that discover other Activities are independent of the fitness function for code coverage. These individuals automatically become the previous Activity transition genes for the new Activity they discover, meanwhile the code coverage fitness function ensures that the objective of maximizing code coverage. As previously mentioned, the discovery of all possible transitions to other Activities from an Activity is encoded in gaining maximal code coverage for that Activity as shown in Figure 7.4.

Unlike Chapter 6, where the transition paths are statically generated, the fitness function here does not consider the number of nodes covered to the next segment or the uniqueness of the traversed path. Attaining full code coverage of the current Activity ensures that all subsequent segments are discovered.

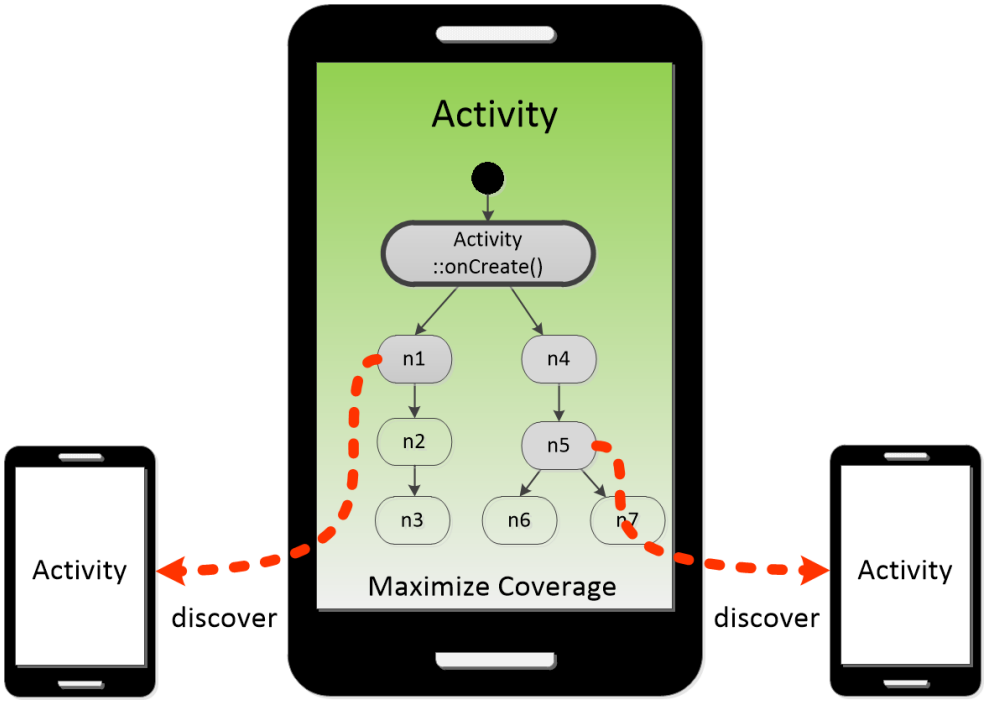


Figure 7.4: Fitness evaluation. Discovery of other Activities is implicitly encoded in maximizing code coverage in current Activity.

Chapter 8: Cloud Testing Framework

The unprecedented power of cloud computing is widely used in both academia and industry. Since evolutionary testing requires a large number of test cases to be executed, there are often performance concerns. I use the computational power of cloud computing to mitigate the performance and scalability issues.

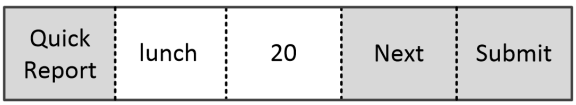
8.1 Test Case Generation

Both approaches from Chapter 6 and Chapter 7 generate test cases in the Robotium [29] format. Robotium is an Android test framework that automates the user interactions with apps via the *Solo* abstraction. It behaves as the user and supports test cases that span multiple Android Activities making it ideal for segmented evolutionary testing. Figure 8.1 shows a sample test case for the ERS application from Chapter 4.1.

The sample test case shows the *quick report* use case which has three segments. We can see that the genes from each segment are composed together to form the entire use case. The genes from an individual are directly mapped to Robotium actions and are executed from the left most gene to the right most gene. However, the test case is fully executable at the end of each segment. In each segment, the evolutionary process finds the sequence of events and inputs to reach the next segment.

The format for the test cases are identical in both the static and dynamic approaches. The tests are also executed the same way and the coverage and crash information is collected the same way. In each generation of both of the algorithms, all individuals are used to generate test cases similar to Figure 8.1, and assessed for fitness based on the coverage and transition information. Since the evolutionary process is repeated per segment, I use cloud computing to alleviate the scalability concerns.

(a)



(b)

```
package com.example.template.test;

import android.test.ActivityInstrumentationTestCase2;
import com.robotium.solo.Solo;
import edu.gmu.ezexpensereport.ERTypeActivity;

public class SampleTest extends
ActivityInstrumentationTestCase2<ERTypeActivity>
{
   private Solo solo;

   public SampleTest()   {
      super(ERTypeActivity.class);
   };

   public void setUp() throws Exception   {
      solo = new Solo(getInstrumentation(), getActivity());
   };

   //@Smoke
   public void atestMethod() throws Exception   {

   //Segment 1 - Expense Report Type Activity
   solo.clickOnButton("Quick Report");

   //Segment 2 - Expense Item Activity
   solo.enterText(0, "lunch");
   solo.enterText(1, "20");
   solo.clickOnButton("Next");

   //Segment 3 - Report Summary Activity
   solo.clickOnButton("Submit");

   };

   @Override
   public void tearDown() throws Exception {
      solo.finishOpenedActivities();
   };

};
```

Figure 8.1: (a) An individual for the Quick Report use case in the ERS, and (b) sample Robotium test case corresponding to the individual.

8.2 Amazon Web Services

Amazon Web Services (AWS) [3] is a cloud computing services provider with various offerings. It provides virtual servers on demand, storage on demand, databases, networking, identify management, analytics, as well as deployment services. These services are primarily categorized into Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS).

These service resources are elastic and can be scaled up or down in real time. These services are managed by using an interactive web interface. Amazon also provides a Command Line Interface (CLI) [2] to provision new servers, storage and services. By using the CLI, it is possible to automate the same functionality provided by the web interface.

8.3 Cloud Test Manager

Traditional evolutionary testing requires a large number of individuals or test cases to be evaluated for fitness. In the case of segmented evolutionary testing, the number of individuals that need to be evaluated increases by the number of segments due to repeating the evolutionary process in each segment.

Typically test cases are executed using an emulator or real device for Android. The Android emulator is known to be slow, thus evaluating test cases for fitness in sequence is not feasible when executing large number of test cases.

As depicted in Figure 8.2, I developed a Cloud Test Manager (CTM) framework that uses the AWS API to provision virtual servers and execute test cases in parallel. The framework is designed to naturally work with EvoDroid, but also for general purpose parallel execution of test cases.

Either EvoDroid or other applications submit an app along with a set of test cases to the CTM. Execution parameters, that specify the number of parallel processors, are also supplied. The app binary along with the test cases are stored in the Test Case Repository (TCR). A manually configured virtual machine, pre-configured with the Java, Android

emulator etc., is created and the CTM application is installed on it. This CTM application on the server is marked as the *coordinator*.

The coordinator monitors the TCR for any new apps and/or test cases that is marked ready for execution. The CTR application launches instances using the AWS API and replicates itself based on parameters provided by EvoDroid or other callers. The number of instances for each app forms a sub cloud cluster as shown in Figure 8.2. The newly replicated launched instances also contain the CTM application, but with a *processor* configuration. Upon initialization, the coordinators register themselves with the TCR and begin processing tests that are marked ready for execution. The application under test is initially copied into a local temporary location. If the application has been flagged as updated, the local copy is destroyed and the latest is retrieved from the TCR.

As each *processor* virtual machine executes the tests on the application under test, it collects coverage information as well as failure information. These outputs are saved back into the TCR. Each *processor* virtual machine periodically checks the TCR until all test cases have been executed. The *coordinator* virtual machine monitors if all tests for a given application has been run and shuts down the *processor* virtual machines as needed.

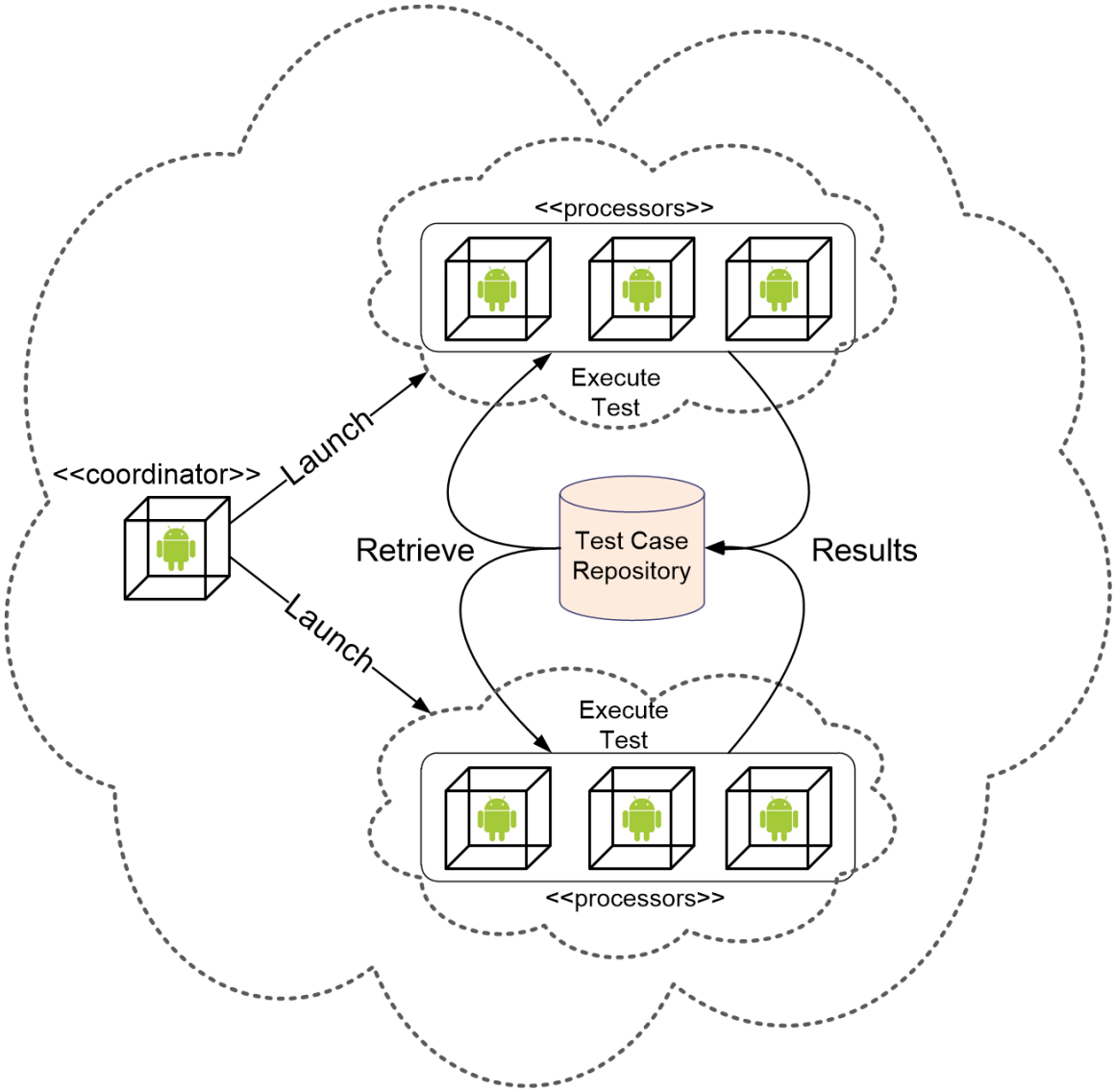


Figure 8.2: Test execution cloud infrastructure. Cloud Test Manager application is deployed on each node as a *coordinator* or a *processor*.

Chapter 9: Evaluation Subject Generation

The research community has responded with an array of testing tools and techniques to tackle the burgeoning app testing challenge [35, 36, 42, 55, 76, 87]. Many of the techniques provide tools that use full or partial automation in an attempt to tackle the scale of the problem.

One of the challenges faced by the research community is not being able to locate consistent and comparable evaluation subjects and not being able to run experiments on such subjects in a controlled setting. Just as no testing tools are identical, no two apps are identical in terms of complexity or suitability as test subjects. There is a lack of consensus on the complexity of apps as complexity is extremely difficult to assess in general. Source lines of code (SLOC) is a popular metric that is typically used in experiments due to its simplicity. However, it is rarely an accurate representation of the complexity of an app. This is especially true in the case of Android as the apps are built on top of an Application Development Framework (ADF) that provides rich out of the box capabilities. As such, it is possible to write very complex apps in Android with just a few lines of code. There is a need to identify the properties that contribute to complexity of apps; and a way to control these properties to allow researchers to quickly assess and compare their techniques with varying degrees of complexity.

In my research, I developed DroidGen, a test subject generation tool that aids in benchmarking and evaluating the strengths and weaknesses of the various app testing tools for Android. DroidGen is capable of generating fully functioning synthetic Android apps in a controlled manner.[1] To ensure the synthetic apps are representative of the complexity of real apps, I identified properties pertinent to the complexity of Android apps and conducted

[1]For the purposes of this research, I use the terms *synthetic app*, *generated app*, *test app* and *test subject* interchangeably.

an empirical study, by analyzing approximately 100 apps from an open source app market called F-Droid [19], and collected data on these property metrics. The data gathered from these complexity metrics are used by DroidGen to generate desirable test apps that exhibit one or more of these properties at varying degrees. It should be noted that I am not advocating substituting real apps for synthetic apps when performing evaluations. Rather, I am providing a supplementary tool that is able to create test applications that represent different levels of complexity without having to consider application-specific features. This allows the research community to quickly assess their testing techniques to stay a step ahead in ensuring their tools can handle different aspects of the growing complexity of Android apps.

9.1 Empirical Study

I conducted an empirical study to gather data on properties that pertain to complexity of Android apps. The data collected here is used by DroidGen to generate synthetic apps that are representative of the complexity of real apps on one or more of these properties. The data is meant to serve as a means of potentially benchmarking Android app testing tools by standardizing test subjects.

Setup. The empirical study informs us of interesting information regarding Android apps and provides us with characteristics that can impact the complexity of apps to determine their suitability as test subjects. I used approximately 100 apps from the open-source market F-Droid [19]. Each app was chosen at random in various categories (e.g. Internet, Office, etc.). I ensured that each selected app was compilable and executable.

To analyze apps, I built a custom static analysis tool that leverages MoDisco [25], an open-source program analysis tool, and Metrics [30], an Eclipse plug-in for calculating software metrics (e.g., McCabe Cyclomatic Complexity [92]). My custom static analysis processes the Android manifest file and extracts information about the app's components from it.

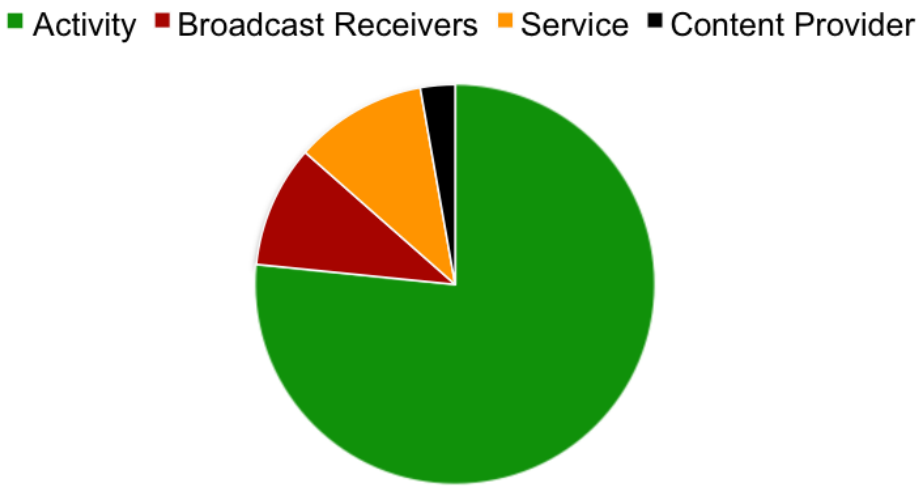


Figure 9.1: Breakdown of Android Components

Recall from Chapter 2 that Android apps have multiple entry points corresponding to different callbacks and events in an Android component lifecycle. Each one of these entry points is the root node of a call graph representing a different execution context to be tested. To capture insights into complexity of these different execution contexts, I used MoDisco [25] to count the Number of Root Nodes (NRN) and the longest Method Call Length (MCL) in the apps.

The Eclipse Metrics [30] plug-in uses object-oriented complexity metrics [74,91] to analyze general Java source code. The metrics collected by this Eclipse plug-in, listed in Table 9.1, contribute to the complexity of Android apps as they are inherently object-oriented.

Findings. I focus on findings regarding the frequency of component usage, the distribution of key complexity metrics, and the statistics of complexity metrics.

My custom static analysis counts the number of major components in Android apps. Figure 9.1 shows the breakdown of the major components. We can see that Android apps heavily use Activities (4.65 on average). Broadcast Receivers and Services are used but not as commonly (.6 and .65 on average). Content providers are used the least often (.17 on

average). The numbers indicate the average occurrence of these components in Android apps.

MoDisco provides us with information regarding the call graphs of apps. Figure 9.2 depicts the distribution of the number of root nodes, i.e., the entry point methods in the call graphs of an app, and the distribution of the length of the longest method-call sequence. These two distributions showcase the effect of the event-driven nature of the Android framework and component lifecycle. We can see that any testing tools and techniques for Android apps need to account for the event-driven nature of Android. This is also due to the fact that most Android apps are inherently GUI-based, corroborated by the number of Activity components shown in Figure 9.1. Therefore, test subjects should, in most scenarios, be GUI-based and contain many disparate call trees to handle the component lifecycle events and GUI events.

The data collected from executing the Metrics plug-in in each of the apps can be seen in Table 9.1. For example, we can see that the average McCabe Cyclomatic Complexity, i.e.,

Table 9.1: Object-Oriented Complexity Metrics

Complexity Metric	Average Value	Standard Deviation
McCabe Cyclomatic Complexity (VG)	2.34	0.67
Number of Parameters (PAR)	1.03	0.32
Nested Block Depth (NBD)	1.62	0.29
Afferent Coupling (CA)	2.54	3.78
Efferent Coupling (CE)	3.50	2.35
Instability (RMI)	0.71	0.24
Abstractness (RMA)	0.05	0.08
Normalized Distance (RMD)	0.26	0.22
Depth of Inheritance Tree (DIT)	1.94	0.38
Weighted Methods per Class (WMC)	7.64	4.67
Number of Children (NSC)	0.11	0.14
Number of Overridden Methods (NORM)	0.25	0.19
Lack of Cohesion Methods (LCOM)	0.13	0.08
Number of Attributes (NOF)	1.72	1.10
Number of Static Attributes (NSF)	5.42	5.55
Number of Methods (NOM)	2.98	1.69
Number of Static Methods (NSM)	0.33	0.39
Specialization Index (SIX)	0.18	0.17
Number of Classes (NOC)	9.52	4.83
Number of Interfaces (NOI)	0.27	0.56
Number of Packages (NOP)	5.27	5.17
Method Lines of Code (MLOC)	9.87	3.80

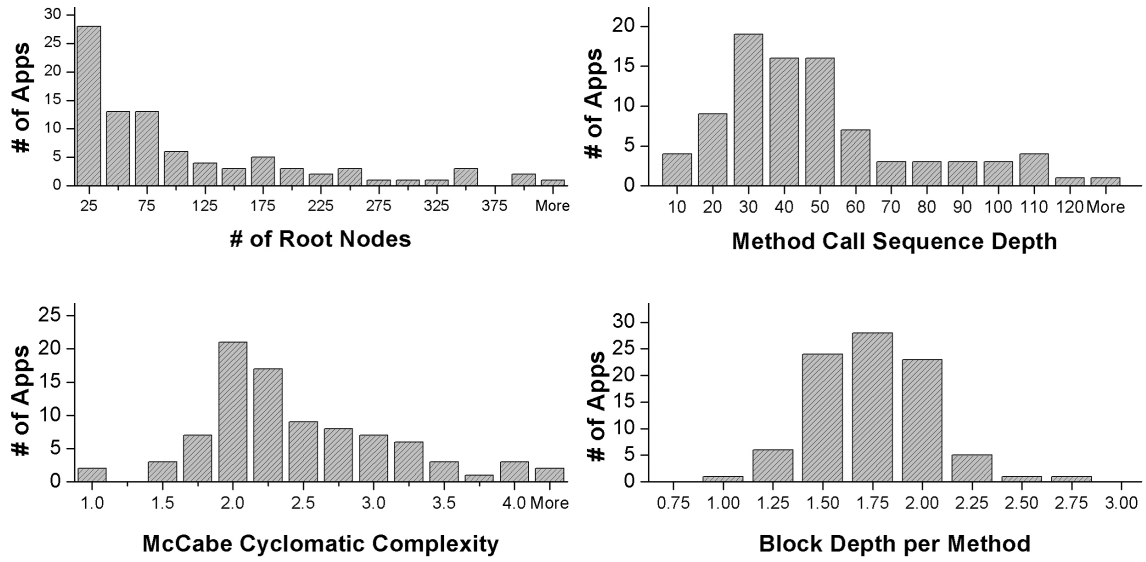


Figure 9.2: Android complexity metrics distribution from a random sample of 100 apps.

the number of control-flow branches per method, in Android apps is 2.34, and on average the Nested Blocked Depth, i.e., the average number of nested condition statements per method, is 1.62. Figure 9.2 depicts the distribution of the McCabe Cyclomatic Complexity, as well as Nested Block Depth. These types of values along with the standard deviation are used by DroidGen when generating constraints in an application.

9.2 DroidGen

DroidGen is a tool that generates fully functional Android apps for the purposes of evaluating, benchmarking, and comparing Android app testing tools. To ensure the generated apps are representative of real world apps, the synthetic test subjects from DroidGen are based on the data collected from the empirical study in Sections 9.1 that represent complexity of the real apps.

As shown in Figure 9.3, DroidGen has three major components that are used to generate

the apps. The Input Parser component parses the input parameters file including the desired complexity settings; then the App Generator component uses the data from the empirical study, a template skeleton app, and Android specifications to generate an app; and finally the Instrumentation component configures the app so it can be instrumented at the class, method, and branch levels. By default, DroidGen executes in a multi-threaded configuration since many apps may be generated. The number of threads to execute along with a temporary working location must be provided.

The Input Parser component takes an XML input parameters file and parses it for the App Generator component. The XML parameter file specifies the list of metrics from Table 9.1 and the desired value range of each metric in the generated test subjects. Each complexity metric is also given a priority with respect to the others for DroidGen to satisfy. Similarly, if some of the metrics are not important to the evaluation at hand, they can optionally be ignored by setting a flag alongside the metric in the parameter file. The parameter file also contains the output location, Android target version, name of the package for the apps, component naming patterns (e.g. appending an incremental number), number of input constraints and ordered event sequence constraints to include, types of instrumentation to include, as well as the number of apps to generate.

The App Generator component uses the data from Section 9.1, different versions of

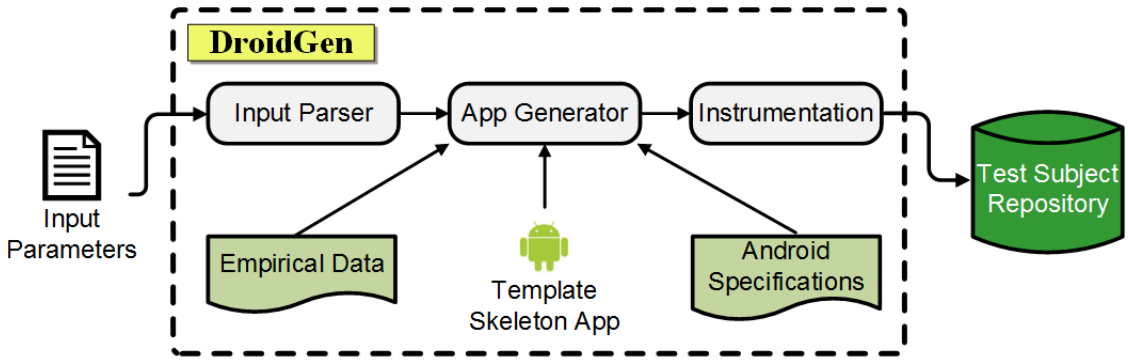


Figure 9.3: Overview of DroidGen

Android specifications to maximize compatibility, and a template skeleton app to generate an Android application on the fly. The template app is manually developed and tested to ensure that it runs properly. The App Generator component supports generating apps that constrain access to a subset of its components based on ordered event sequences of specified lengths, the maximum number of input constraints, maximum number of nested constraints, and the probability of satisfiabilty for the constraints. There are times when the App Generator is not able satisfy all of the parameters. In such instances, it satisfies as many of the metric values as possible based on the priority provided in the input file. The App Generator component starts with the average values from Section 9.1 and tweaks and adjusts the app iteratively until the specified ranges are met. It analyzes the generated app using the tools described in Section 9.1. These are the tools used to do the empirical study and they are used by this component at run-time to check the properties of the generated apps. All major Android components are included with respect to the distribution seen in Figure 9.1. Additional components are added based on input preference and Figure 9.2 to increase or decrease the number of root nodes. Finally, the data from Table 9.1 is used to organize packages, components, classes, and methods and add constraints within the app.

The Instrumentation component statically injects code in the generated apps at the class, method, and branch levels to monitor the coverage achieved by each test. Depending on the configuration options, the generated app can log to a file, database, or simply output to the screen if the instrumentation location is executed. This component gives an easy mechanism for reachability, coverage, and performance instrumentation. It also provides the mechanism for high level data flow instrumentation such as observing input values from the GUI at various execution locations. Once the test subjects are generated and instrumentation is completed, they are deployed to a Test Subject Repository on the local drive, network share, or on the cloud depending on the user preference.

Chapter 10: Evaluation Results

I evaluated EvoDroid on a large number of apps with varying characteristics. The goal of my evaluation was twofold: (1) compare the EvoDroid code coverage against the prior solutions, and (2) characterize its benefits and shortcomings.

10.1 Experiment Environment

Evolutionary testing requires the execution of a large number of tests. This is especially challenging in the case of the Android emulator [7], as it is known to be slow even when running on workstations with the latest processors and abundant memory. To mitigate this issue, I have developed a novel technique to execute the tests in parallel, possibly on the cloud, which makes it suitable for use by small as well as large organizations. The details of this infrastructure is provided in Chapter 8.

I set up an instance of Amazon EC2 virtual server running Windows Server 2012, and configured it with Java SDK, Android SDK, Android Virtual Device, and a custom Cloud Test Manager application as described in Chapter 8. For each test, the test execution manager launches the emulator, installs the app, sets up and executes the test. It is also responsible for persisting all of the results, along with the log and monitored data, to an output repository. A virtual machine image was created from the above instance to be replicated on demand. With this, I was able to scale in near-linear time and cut down on the execution time. I report the results for both extremes: when the test cases are executed in sequence using a single processor, and when they execute completely in parallel.

I have implemented EvoDroid using ECJ [17], a prominent evolutionary computing framework. In the experiments, I used EMMA [18] to monitor for code coverage, and all of the test cases were in Robotium [29] format.

10.2 Experiment Setup

I compare EvoDroid with Android Monkey [10] and Dynodroid [87] in terms of code coverage and execution time. Android Monkey is developed by Google and represents the state-of-the-practice in automated testing of Android apps. It sends random inputs and events to the app under test. Dynodroid [87] is a recently published work from researchers at Georgia Tech that uses a smaller number of inputs and events than Monkey for reaching similar coverage. I am not able to compare directly with EXSYST [71] and EvoSuite [72], as they are not targeted for Android. I do not compare against [113] as that is for model generation only, while EvoDroid creates models and performs a step-wise segmented evolutionary search.

In the evaluation of any stochastic search algorithms, such as EvoDroid, it is desirable to compare the results of the algorithm against the unbiased random sample of the solution space. I believe the comparison against Monkey and Dynodroid helps us in that vein. It should be noted that unlike EvoDroid, Monkey and Dynodroid are not designed to run in a distributed manner, and neither tools are configurable to run for a specific amount of time.

To achieve a fair comparison with Android Monkey and Dynodroid, however, I had to allot each approach similar number of events. The events for Android Monkey and Dynodroid are similar to the genes in EvoDroid. Since there is no one-to-one mapping, I ran EvoDroid first. I then mapped the total number of generated tests in EvoDroid to Monkey and Dynodroid events. Thus, the number of events allotted for running Monkey and Dynodroid varied as a function of the number of test cases executed for EvoDroid as follows: $max(g) \times t$, where g is the maximum number of genes allowed for test cases in EvoDroid and t is the number of test cases executed for a given app.

In all experiment scenarios, for each segment in each path, I used a maximum of 10 generations of 10 individuals with a maximum of 10 genes. The number of maximum generations along with the coverage of all segments served as the terminating conditions. Euler's constant ($e = 2.718$) was used as the crossover decay number in eq. 5.1, and during

the mutation phase each gene had a 20% chance of mutation in EvoDroid.

To evaluate EvoDroid, two sets of experiments were performed. The first on apps developed by independent parties from an open source repository, and the second on synthetic apps. The synthetic apps helped us benchmark EvoDroid's characteristics in a controlled setting.

10.3 Open Source Apps

I selected 10 open source apps to evaluate the line coverage between EvoDroid, Monkey, and Dynodroid. I was not able to run Dynodroid on two of the subject apps, and thus I am not able to report on those. As shown in Figure 10.1, EvoDroid consistently achieve significantly higher coverage than both Monkey and Dynodroid. On average statically extracting the models in EvoDroid achieves 47% and 27% higher coverage than Monkey and Dynodroid, respectively. On average dynamically extracting the models achieves 45% and 25% higher code coverage than Monkey and Dynodroid, respectively. EvoDroid static and dynamic approaches comparatively achieve similar code coverage depending on the quality of the models as fundamentally both algorithms apply a segmented evolutionary approach.

The generation of test oracles is outside the scope of my work, nevertheless I collected information about unhandled exceptions, which allowed us to detect several defects in these apps. For instance, I found several cases of unhandled *number format exception* in Tipster, TippyTipper, and Bites that were due to either leaving the input fields empty, clicking a button that clears the input field followed by clicking a button that would operate on the inputs, or simply putting a string that could not be converted to a number. As another example, I found a defect in Bites, an app for finding and sharing food recipes, in which an unhandled *index out of bounds exception* would be raised when editing recipes without adding the ingredients list first.

Some of the reasons for not achieving complete coverage are unsupported emulator functions, such as *camera*, as well as spawning asynchronous tasks that may fail, not finish by the time the test finishes, and thus not get included in the coverage results. Other reasons

include code for handling external events, such as receiving a text message, dependence on other apps, such as calendars and contacts lists, and custom exception classes that are not encountered or thrown. Additionally, some of the applications contained dead code or test code that was not reachable, thus the generated static EvoDroid models would not be fully connected or the dynamic approach would not be able to reach a particular segment at run-time. Indeed, in many of these apps achieving 100% coverage is not possible, regardless of the technique.

The limitations of emulators, peculiarities in the third-party apps, and incomplete models made it very difficult to assess the characteristics of EvoDroid independently. The static models generated by EvoDroid are sometimes incomplete due to Android version fragmentation, variability of programming styles, third party libraries, code generation via reflection, and anonymous class declarations among other reasons. While the dynamic approach does

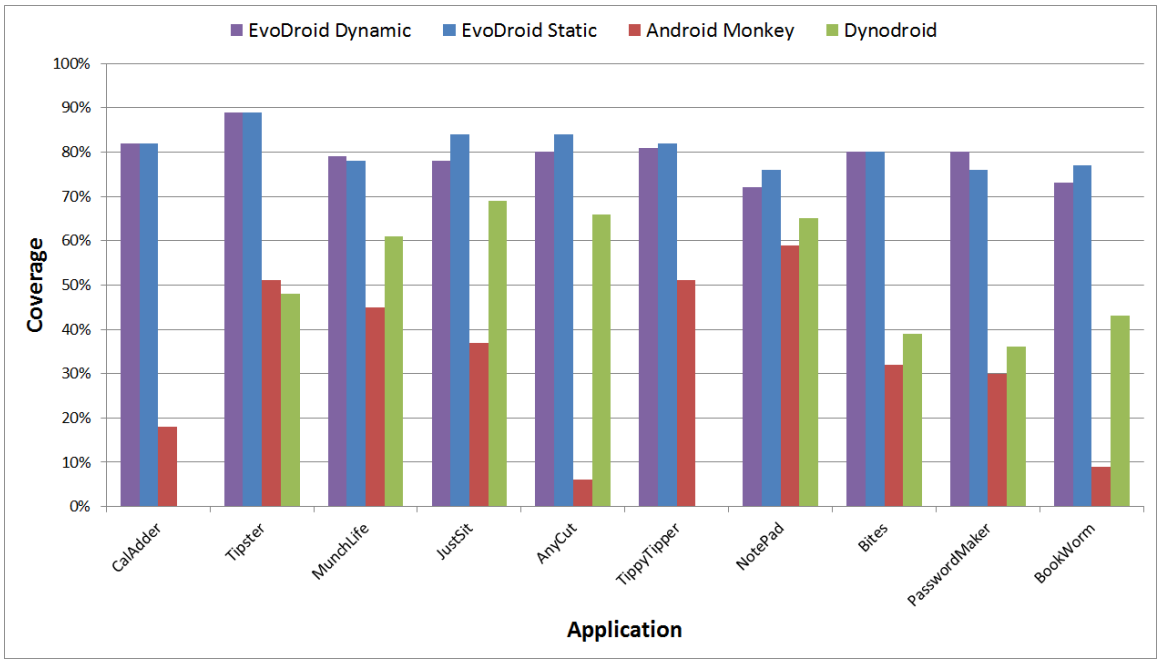


Figure 10.1: Line coverage results for testing apps from different complexity classes.

not suffer from the same limitations because it does not use static analysis, its models are also sometimes not complete. This is partially due to not being able to discover and crawl to other nodes as constraint values need to be solved to reach those nodes.

In other words, it was not clear whether the observed accuracy and performance was due to the aforementioned issues, and thus an orthogonal concern, or due to the fundamental limitations in EvoDroid's approach to test generation. I, therefore, complemented the evaluation on real apps with a benchmark using synthetic apps, as discussed next.

10.4 Synthetic Benchmark Apps

To control the characteristics of the subjects (i.e., apps under test), I developed an Android app generator that synthesizes apps with different levels of complexity for the experiments as described in Chapter 9. Since I needed a way of ensuring the synthetic apps were representative of real apps, I first conducted an empirical study involving 100 real world apps chosen randomly from an open source repository, called F-Droid [19]. The selected apps were in various categories, such as education, Internet, games, etc. I analyzed these apps according to four complexity metrics that could impact EvoDroid:

- *Root Nodes per App* — the number of disconnected call graphs in the app; these are the methods called by ADF, and potentially the break points for segments.
- *Method Call Sequence Depth* — the longest method call sequence in the app.
- *McCabe Cyclomatic Complexity* — the average number of control flow branches per method.
- *Block Depth per Method* — the average number of nested condition statements per method.

Figure 9.2 shows the distribution of these metrics among the 100 Android apps from F-Droid. My app generator is able to synthesize apps with varying values in these four metrics. Since I wanted to evaluate the accuracy and performance of EvoDroid on subjects

with different levels of complexity, I had to derive some complexity classes from this data. For that, I aggregated the data collected through the empirical study, as shown in Figure 9.2, and divided it into 9 equal complexity classes, ranging from 1 to 9. For instance, the 1st complexity class corresponds to the 10th percentile in all of the four metrics shown in Figure 9.2. Essentially an app belonging to a lower class is less complex with respect to all four metrics than an app from a higher class.

10.4.1 Impact of Complexity

To benchmark the impact of complexity on EvoDroid, I generated two apps for each complexity class. Apps were set up such that exactly 1 path contained no input constraints, while other paths contained nested conditional input constraints. These constraints were generated to simulate the *Block Depth per Method* dimension, and would have to be satisfied in order for the search to progress further and attain deeper coverage. The generated conditional statements had a 50% satisfiability probability given a random input value. Of course, some of the conditional statements were nested in the synthetic apps, resulting in a lower probability of satisfying certain paths.

The line coverage results are summarized in Figure 10.2. As the complexity class of apps increases, the coverage for Monkey and Dynodroid drops significantly. Since EvoDroid logically divides an app into segments, the complexity stays relatively the same, i.e., it is not compounded per segment. In all experiments, EvoDroid achieved over 98% line coverage. The cases where 100% coverage is not reached is due to the algorithms abandoning the search when reaching the maximum number of allowable generations. Increasing the number of generations is likely to resolve those situations.

Once Monkey traverses a path, it does not backtrack or use any other systematic way to test the app. Therefore, Monkey's test coverage is shallow as others have confirmed in [87]. Dynodroid periodically restarts from the beginning of the app, and is able to outperform Monkey. Note that for very complex apps, Dynodroid would crash, and thus I was not able

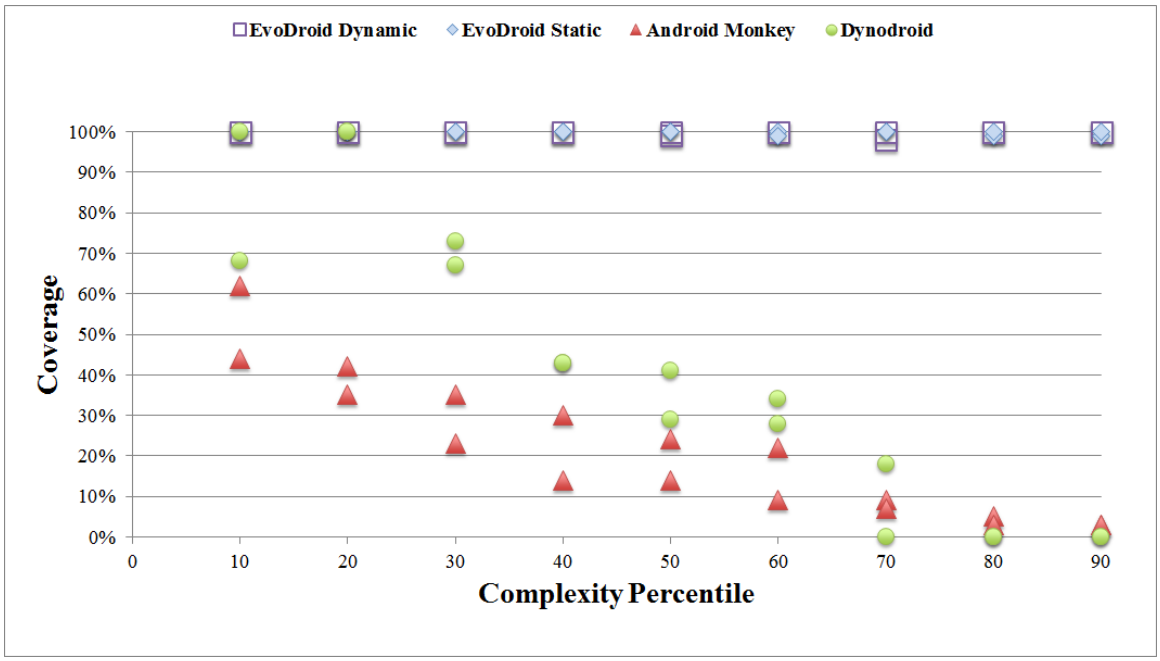


Figure 10.2: Line coverage results for testing apps from different complexity classes.

to obtain results.[1]

Table 10.1 summarizes the execution time for EvoDroid, Monkey, and Dynodroid. Even though the execution time for EvoDroid significantly increases as the apps become more complex, it could be alleviated by running them in parallel, possibly on the cloud. The parallel times presented are roughly equivalent to the worst path execution time for both algorithms. The numbers presented assume as many parallel instances running as there are test cases. In practice, we expect EvoDroid to be executed on several machines, but perhaps not hundreds, producing an execution time in between the worst case and best case reported in Table 10.1. We can see that as the depth of the segments increases, the time to execute EvoDroid increases. The results also show that Monkey runs fairly quickly, while Dynodroid takes longer, as one would expect due to its backtracking feature.

[1]This is because above 10,000 events Dynodroid needs more than 3.4GB memory, which is more than the maximum memory size the 32-bit virtual machine, that the Georgia Tech researchers provided me for the experimentation, could support.

Table 10.1: Execution time for testing apps from different complexity classes (in minutes).

Complexity Class	# of Segments	# of Test Cases	Monkey/ DynoDroid Events	EvoDroid Dynamic Single CPU	EvoDroid Dynamic Parallel	EvoDroid Static Single CPU	EvoDroid Static Parallel	Android Monkey	DynoDroid
1	4	30	300	38.06	1.59	41.55	1.65	1.52	235.57
1	6	40	400	43.23	1.68	44.10	1.75	1.55	273.27
2	8	50	500	65.60	1.96	52.00	2.01	1.52	309.43
2	9	60	600	76.54	2.10	88.31	2.20	1.53	325.78
3	17	130	300	280.47	2.53	258.77	2.40	1.65	458.90
3	19	140	140	274.80	2.39	293.38	2.53	1.58	407.60
4	25	210	2100	465.87	3.11	686.47	3.31	1.97	1091.00
4	33	220	2200	430.50	3.41	388.30	3.11	1.72	1073.40
5	36	260	2600	656.33	3.89	796.23	4.13	1.88	592.27
5	48	290	2900	702.43	4.12	692.77	3.92	2.23	823.57
6	49	280	2800	899.86	4.79	1107.79	4.90	2.02	623.57
6	47	520	5200	1297.32	5.39	957.97	4.43	2.13	1100.70
7	109	750	7500	2455.01	5.01	2725.27	5.11	2.50	1233.90
7	110	1110	11100	3288.67	5.67	2999.87	5.53	2.82	-
8	233	1800	18000	7547.19	6.58	7645.32	6.20	4.03	-
8	282	1960	19600	7815.30	6.46	8035.17	6.53	4.17	-
9	345	3640	36400	17543.75	6.99	13713.25	6.81	5.43	-
9	487	3680	36800	19465.10	7.20	21395.50	7.59	6.00	-

10.4.2 Impact of Constraints

Input constraint satisfaction is a known weakness of search based testing techniques. A set of experiments was conducted to assess the efficacy of my approach as the satisfiability probability of conditional statements was lowered below 50%. I took the second app from the 3rd complexity class (shown in Figure 10.2) and lowered the satisfiability probability of its conditional statements to 25%, 10%, and 1%. As shown in Figure 10.3, when the probability of constraint satisfiability decreases, the line coverage drops significantly for EvoDroid. Android Monkey coverage stays the same as it takes the one path with no constraints and does not backtrack. The coverage for Dynodroid drops also, but remains better than Monkey, as it restarts from the beginning several times during execution.

The results demonstrate that EvoDroid (as well as any other evolutionary testing approach) performs poorly in cases where the apps are highly constrained (e.g., the probability of satisfying many conditional constraints with random inputs is close to zero, such as an *if* condition that *specifies* an input value to be *equal* to a *specific value*). Fully addressing

this limitation requires an effective approach for solving the constraints, such as symbolic execution, as described further in Chapter 12. Fortunately, from Figure 9.2 we see that for a typical Android app, the average cyclomatic complexity is approximately 2.2 and block depth is approximately 1.75. These numbers are encouraging, as they show that on average most Android apps are not very constrained.

10.4.3 Impact of Sequences

Given the event driven nature of Android apps, there are situations when certain sequences of events must precede others or certain number of events must occur to execute a part of the code. I evaluated EvoDroid for these types of situations by generating apps from the 3rd complexity class with ordered sequence lengths ranging from 1 to 5. Sequences of events with these lengths would have to be satisfied, per segment in all paths, in order to proceed with the search (e.g. certain buttons on an Activity must be clicked in a certain

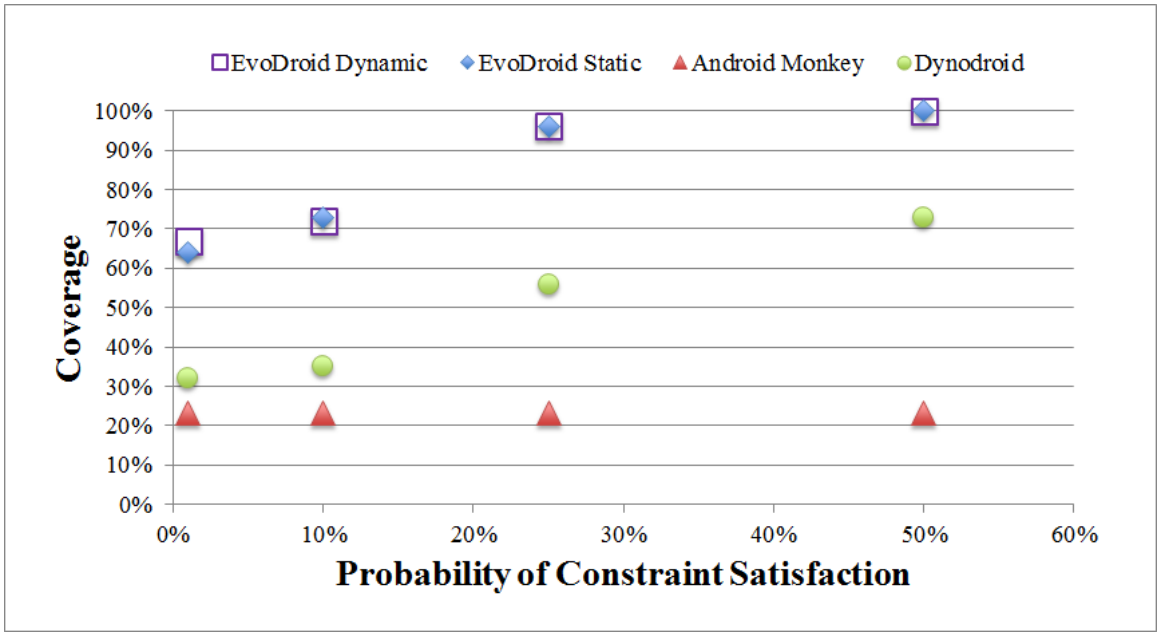


Figure 10.3: Impact of constraints on line coverage.

sequence of length 1 to 5).

Figure 10.4 summarizes the results from these experiments. While EvoDroid's coverage decreases, it does so at a much slower pace than Monkey or Dynodroid. We observe that EvoDroid is effective in generating system tests for Android apps involving complex sequence of events. This is indeed one of the strengths of both evolutionary approaches that is quite important for Android apps as they are innately event driven.

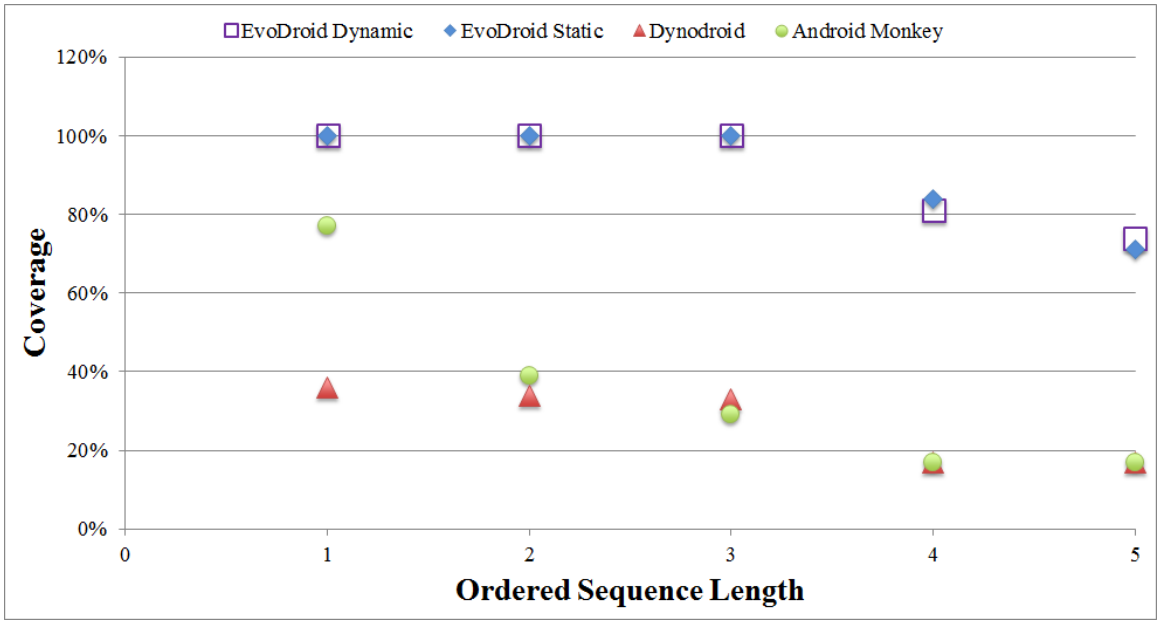


Figure 10.4: Impact of sequences of events on line coverage.

Chapter 11: Applications of EvoDroid

By overcoming the crossover issue in traditional evolutionary testing, EvoDroid fundamentally attempts to address the *reachability* issue. It is often desirable to reach and execute specific locations in applications. EvoDroid's current objective subsumes this problem as it targets and attempts to gain coverage of the entire application. By changing the fitness function in EvoDroid, it is possible to target areas of interest in applications. This chapter outlines some of these use cases.

11.1 Security Vulnerability Testing

EvoDroid can be used for targeted security vulnerability testing as shown in Figure 11.1. We can statically analyze the code looking for well-known vulnerability patterns in the app and label the vulnerabilities found in this way as targets, i.e., locations in the code that are likely to be vulnerable. Afterwards, we can generate candidate exploits (test cases) that consist of inputs and sequence of actions to reach the targets. There are many different types of static analysis techniques for finding vulnerabilities, including Control Flow, Data Flow, and Content analysis. The rule set for these techniques are available from prior research [58] and commercial tools such as Fortify SCA.

EvoDroid is able to complement the results of static analysis with targeted dynamic analysis to produce exploits. Targeted dynamic analysis allows us to verify whether a given vulnerability is exploitable by executing the code. Consider that by simply executing a line of code, we are able to determine unequivocally whether the vulnerability is exploitable or not. The key challenge, however, is finding a test case that would execute a given target.

Code-rewriting techniques can be used instrument the application software under test based on the vulnerability information. For this purpose, code needs to be inserted to

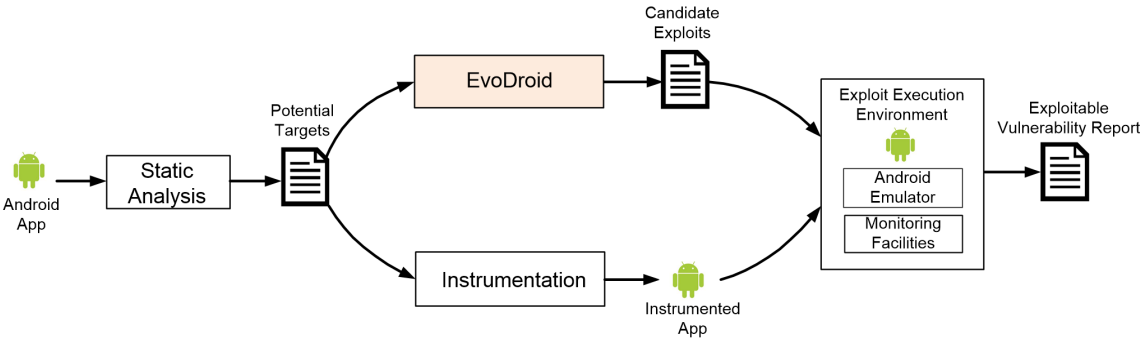


Figure 11.1: Targeted Security Testing

monitor the program status and test results. The process of inserting monitoring code in existing code is called instrumentation. Instrumentation for exploitation monitoring is essentially the same as inserting test oracles in an application. Automated creation of a precise oracle is challenging [106] when the test results should be checked against certain expected output values, rather than the more easily observable phenomena, such as a crash. Additionally, for precise detection of exploits, instrumentation code may have to be customized depending on the vulnerability patterns that are being tested. Each vulnerability pattern has to be monitored and detected in a different way, as each may manifest itself in a different way. EvoDroid can simply focus on the coverage or crash based instrumentation. Specific vulnerability semantic based test oracle generation is not within the scope EvoDroid.

11.2 Incremental Regression Testing

Software apps are typically built by a team of engineers following a predefined software development process or methodology. Most modern approaches and methodologies include test automation and continuous integration testing.

EvoDroid can complement the typical unit tests, that are manually written by engineers, by running during continuous integration testing on a periodic basis. By identifying the

locations of the all of the changed source files using the API from the code repository (e.g. Subversion [31]). Once the added or modified locations have been identified, they serve as the target locations. The approach is shown in Figure 11.2.

This adds significant value to the development team, as the defects that are inadvertently added to newly developed or updated components can be automatically discovered by EvoDroid.

11.3 Field Failure Replication Testing

Manually creating tests to replicate faults submitted from the field is very costly. EvoDroid is able to automatically create test cases for faults that are found in the field as shown in Figure 11.3.

The reports are automatically collected into a crash or fault repository. EvoDroid pulls from this repository on a periodic basis and automatically generate tests cases to reach the locations of the faults.

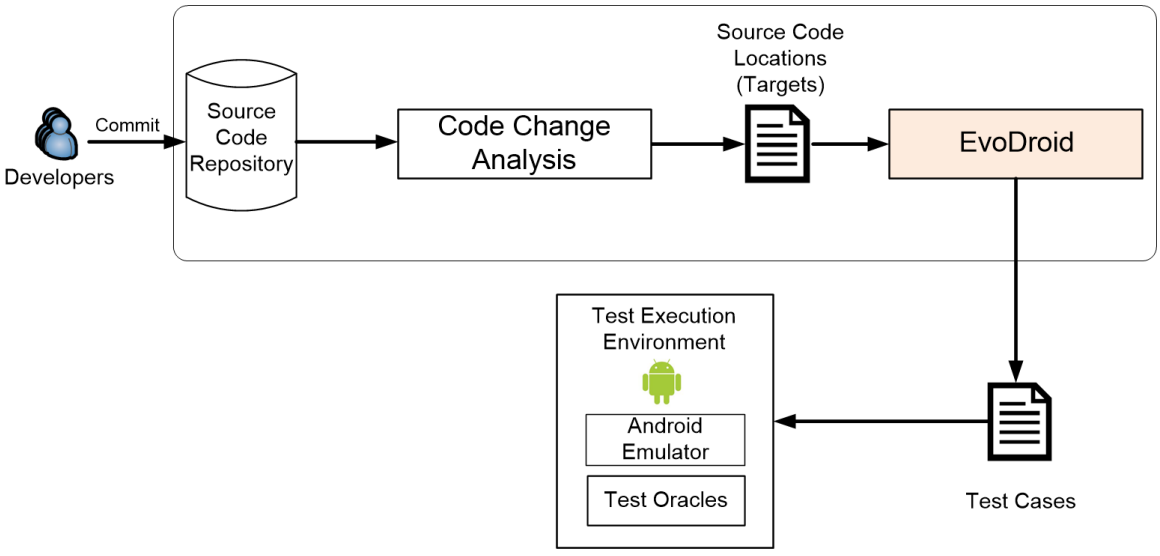


Figure 11.2: Targeted Incremental Testing

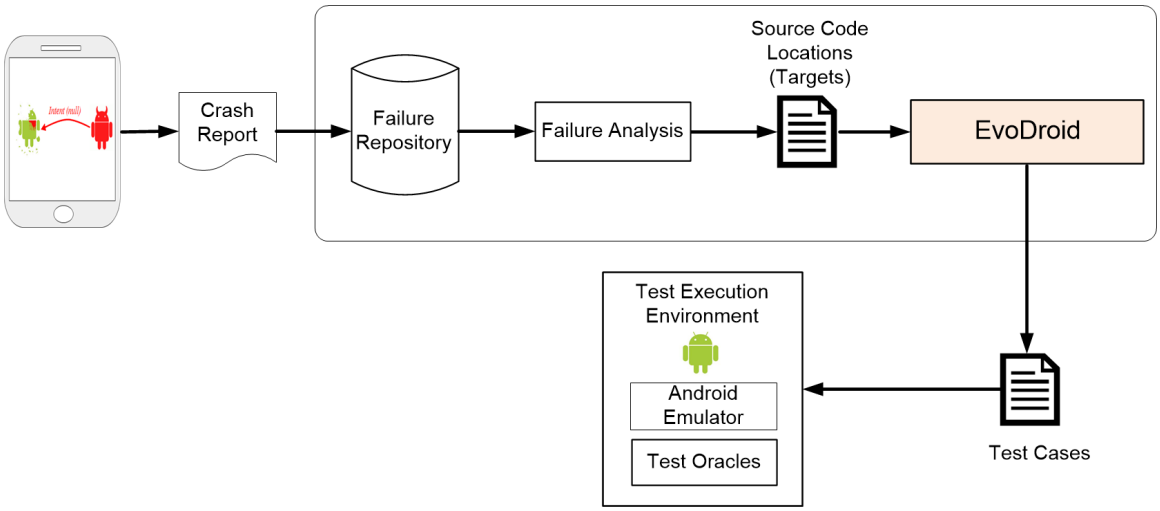


Figure 11.3: Field Failure Testing

For example, if a *null* memory reference fault takes place in the field, the data can be collected with the location of the fault. A test case can automatically be generated to reach the location as a target. This saves a tremendous amounts of debugging time especially as the number of field failures increase.

11.4 Limitations

Using EvoDroid for targeting testing is hindered by the need to solve input constraints. One of the weaknesses of search based testing techniques, such as EvoDroid, is its inability to systematically reason about input constraints. This is especially true as the probability of constraints satisfaction reaches near zero. Symbolic execution has been used by the research community to address this issue. I employ a similar approach in solving input constraints for Android apps as described in Chapter 12.

Chapter 12: Reasoning About Input Values

As seen from the evaluations in Chapter 10, the strength of search based testing techniques, such as evolutionary testing, is reasoning about events. It does not lend itself well when reasoning about input values and constraints. In my research, I explored an input generation solution, called SIG-Droid, that can be used to mitigate this weakness using symbolic execution. This chapter provides implementation details about SIG-Droid.

12.1 SIG-Droid

System Input Generation for Android apps (SIG-Droid) [96] is an input generation and constraint solver tool for Android. It generates proper values for GUI data widgets that take user inputs. The input domain can be quite large, 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 $\frac{1}{10^5}$ chance to reach the state satisfying that condition.

SIG-Droid combines program analysis techniques with symbolic execution [83] to systematically generate inputs for Android apps. To prune the domain of data inputs, SIG-Droid employs symbolic execution, a promising automated testing technique that can effectively deal with constraints. Symbolic execution uses symbolic values, rather than actual values, as program inputs. It gathers the constraints on input values along each path of a program execution, and with the help of a constraint solver generates actual inputs for all reachable paths. While symbolic execution has proven to be effective for unit level testing, there are some unique challenges when testing Android apps.

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 causing 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 [38]. Furthermore, although Android apps are developed in Java, they run on Dalvik Virtual Machine (DVM) [16], 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.

SIG-Droid uses extracted models to exhaustively pinpoint possible ways an app can receive inputs. It 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 *Behavior Model* called *Drivers*. Furthermore, to enable symbolic execution engine to run the apps in the 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.

12.2 Symbolic Execution

Symbolic Execution [83] is a program analysis technique that uses symbolic values, rather than actual values as program inputs. 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 which 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, a simple Java program is depicted in Figure 12.1b, where S0, S1, S2, and S3 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 12.1b 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 12.1b 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 "$X > 0$ & $X \leq 3$ & $Y = X \times 3$", the following two values are obtained "$X = 3$" and "$Y = 9$", which result in taking the bold path in Figure 12.1b and executing S0 and S2. Similarly, using symbolic execution, it's possible to generate all inputs for test methods 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 assertion violation (path 3).

Symbolic Pathfinder (SPF) [32] is a symbolic execution engine for Java programs. It is built on top of Java Pathfinder (JPF) [22], 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 modied JVM [22]. 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 [32]. SIG-Droid extends SPF to support Android apps. By addressing SPF limitations in dealing with event driven nature of Android, it is able to generate inputs and test cases for Android programs.

12.3 Illustrative Example

For illustrating the approach, I use a mobile banking app as a running example. Figure 12.2a shows two of the screens comprising this app: `MainActivity` and `TransferActivity`. The `MainActivity` is the first screen that the user sees when the app is launched. It allows the user to work with her checking or savings account (e.g., see the details of transactions occurring in each account). The `TransferActivity` screen allows the user to transfer money between the checking and savings accounts.

Figure 12.2b shows code snippets realizing one of the functionality provided by this app.

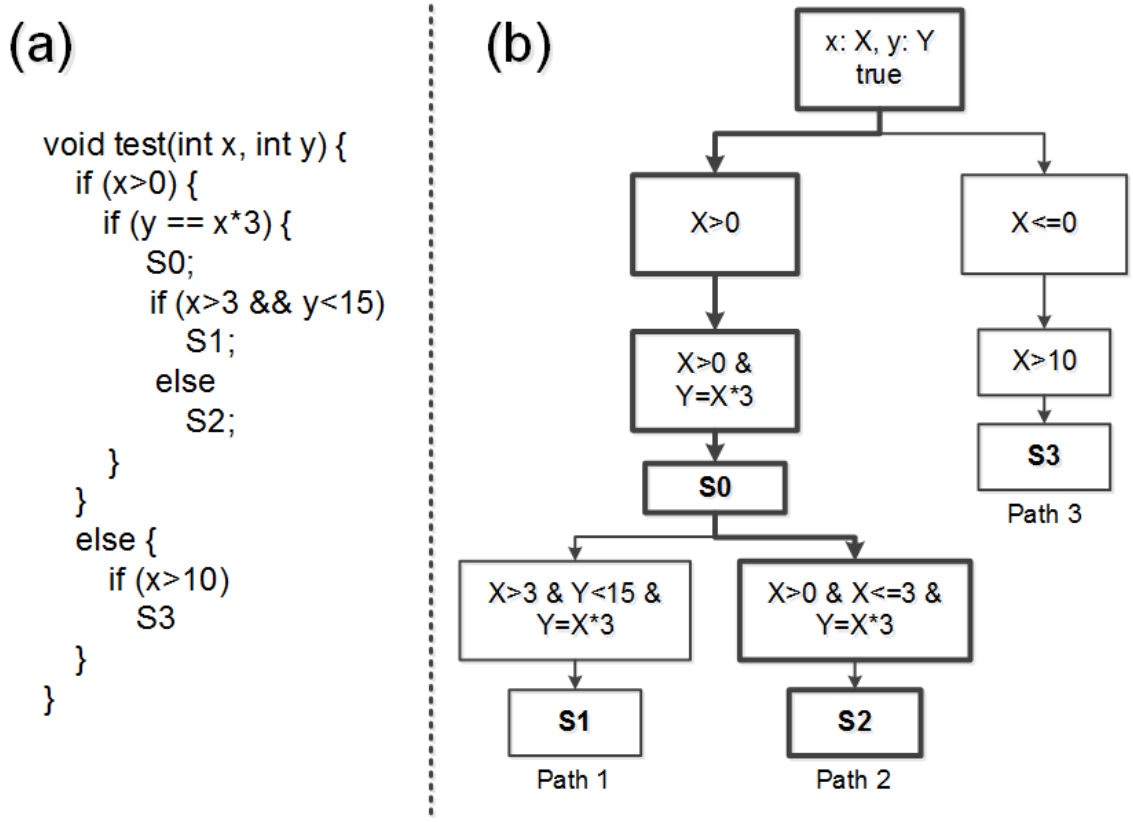
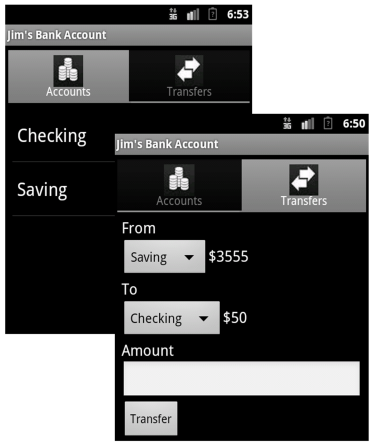


Figure 12.1: Symbolic execution: (a) sample code, and (b) the corresponding symbolic execution tree, where X and Y are the symbolic representations of variables x and y

When the Transfer button (see Figure 12.2a) is clicked, the `onClick` method is called by the ADF. Subsequently, if the transfer amount is less than $5,000, an Intent is sent to the

(a)



(b)

```
public class TransferActivity extends Activity {
   ...
   public class Transfer implements  OnClickListener {
        public void onClick (View v) {
         ...
         EditText amount =
                (EditText)findViewById(R.id.amountId);
        amountValue = Integer.parseInt(
                          amount.getText().toString());
         if (amountValue <= 5000 ){
           ...
            Intent intent = new Intent(
           this,MainActivity.class);
             intent.putExtra("amount", amountValue);
             intent.putExtra("source",srcAccount);
             intent.putExtra("destination",dstAccount);
             startActivity(intent);
         }
         else
           throw new TransferException("Limit Exceeded")
       }
  }
}

public class MainActivity extends Activity {
  public void onCreate (Bundle savedInstanceState) {
     ...
     Intent intent = getIntent();
     int amount = intent.getExtra("amount");
      sourceBalance -= amount;
      destinationBalance += amount;
      startActivity(intent);
    }
  ...
}
```

(c)

```
<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/transferBtn"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Transfer" />
    ...
</LinearLayout>
```

Figure 12.2: Banking App: (a) Screenshots, (b) code snippet from TransferActivity, and (c) snippet from Transfer.xml layout.

`MainActivity` including as payload the transfer amount, source and destination accounts. Finally, `MainActivity` updates the balance in each account to reflect the transfer amount, and displays the updated result to the user.

As previously mentioned, the widgets on each activity are defined in an XML layout file. Figure 12.2c presents a snippet of the layout file for `TransferActivity`.

One conceivable test case generated using one of the existing techniques is clicking on the Transfers tab (thus bringing up the screen corresponding to `TransferActivity`), entering a random value as the amount to be transferred, and clicking on Transfer button. But considering the constraint in the code that the transfer amount cannot be more than $5,000, there is no systematic way of generating tests that cover both possible paths following the constraint. SIG-Droid symbolically executes the parts of the code corresponding to the sequences of events and generates test cases that cover both paths.

12.4 Overview of SIG-Droid

Figure 12.3 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 Behavior Model (BM) represents the event-driven behavior of the app, including the relationships among the event generators and handlers. SIG-Droid uses the *BM* to generate possible use cases of the system (sequence of events), known as *Drivers* in the symbolic execution literature [32].

- The Interface Model (IM) provides a representation of an app's external interfaces and in particular ways in which it can be exercised, e.g., the inputs and events that are available on various screens to generate test cases that are valid for those screens. SIG-Droid uses the *IM* to determine the candidate input values that should be exchanged with symbolic values.

The second component of SIG-Droid is the Symbolic Execution Engine. As mentioned in Section 12.2, 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 *BM* 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

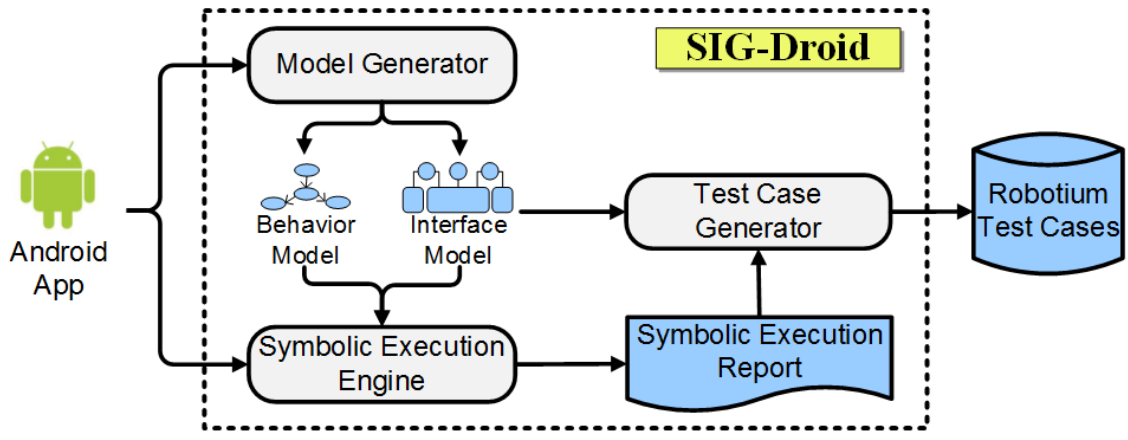


Figure 12.3: High level overview of SIG-Droid.

executed on top of Robotium [29], which is an Android test bed.

The next three sections describe the three components of SIG-Droid in more detail.

12.5 Model Generation

Model Generator extracts two models for each app: *Behavior Model (BM)* and *Interface Model (IM)*.

12.5.1 Behavior Model

The *BM* represents a high-level behavior of the app in terms of the interactions among the event generators and handlers. This model is obtained in three steps: (1) reverse engineering of the app's call graph, which only contains the explicit method calls, (2) using knowledge of the ADF specification to augment the call graph with implicit calls (i.e., event exchange), and (3) pruning the call graph from nodes irrelevant to understanding the event-driven behavior of the app.

The first step of SIG-Droid entails using MoDisco [25], an open source program analysis tool, to extract the app's call graph. Unlike traditional Java programs, Android apps do not contain a main class that becomes the root node of the call graph, where the program is always initiated. Android apps are event driven, meaning that the thread of execution constantly changes context between the application logic, ADF, and user interface. Therefore, instead of a connected call graph that represents the connected set of possible method calls, an Android app is composed of a set of disconnected sub-call graphs that collectively represent the app's logic. These sub-call graphs correspond to all the ways in which an app can be initiated and accessed by the user or the Android platform.

Figure 12.4a shows a subset of the Banking app's call graph obtained from its source code (as described later in this section, the red dashed lines are inferred to create a fully connected graph by extending MoDisco). Boxes in Figure 12.4a represent the methods, and the lines represent the sequence of invocations.

The second step of SIG-Droid is to relate the reverse engineered sub-call graphs to one another, and thus discover the dotted red arrows shown Figure 12.4a. Note that the links between the subcall graphs are implicit (i.e., these calls are initiated by the Android ADF itself) and not recoverable through simple source code analysis tools, such as MoDisco.

We observe that the root node of each sub-call graph is a method call never explicitly invoked from other parts of the application. There are two types of root nodes:

1. *Inter-component root nodes* represent methods in a component that handle events

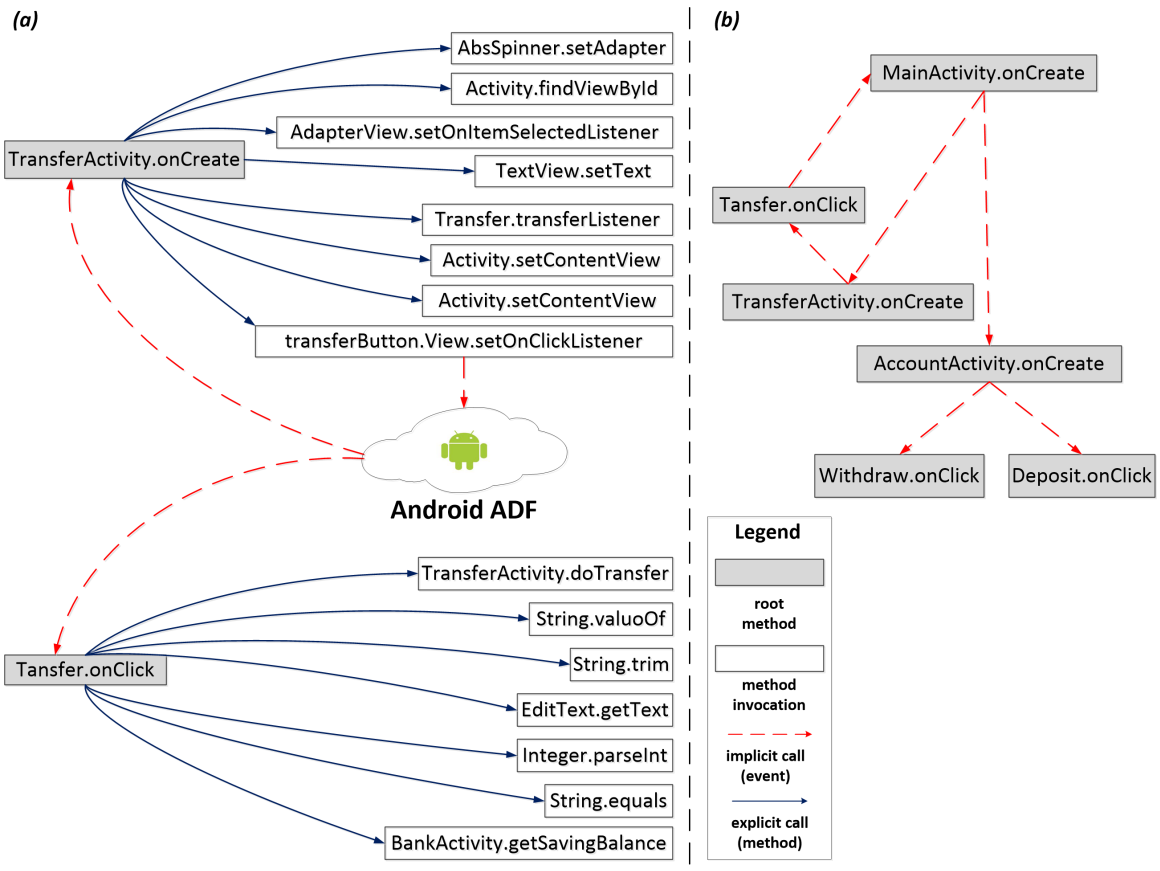


Figure 12.4: (a) Examples of two sub-call graphs automatically inferred for the banking app; augmented red lines represent implicit calls used to connect sub-graphs; (b) the *BM* for the banking app automatically generated after pruning sub-graphs.

generated by other components or Android framework itself. For instance, in the example of Figure 12.4b, `TransferActivity` component sends a `startActivity` event that results in `MainActivity`'s `onCreate()` method to be called.

2. *Intra-component root nodes* correspond to events that are internal to a component. For instance, in the example of Figure 12.4b, when a Button belonging to `TransferActivity` is clicked, the event is handled by the Transfer class within the same activity that implements the `OnClickListener` interface and overrides the `onClick()` method.

In order to resolve all of the implicit links in the app, SIG-Droid traverses the call graph starting with the `onCreate()` root node of the main activity (the starting point of the app). To link the different sub-call graphs, it continues down the graph and identify the leaf nodes where implicit method calls 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 to other root nodes. For example, in Figure 12.2b there is an implicit call from `startActivity` in `TransferActivity` to `MainActivity`'s `onCreate()`. Algorithm 3 shows how implicit calls are extracted, given a set of disconnected call-graphs and the set of caller methods that initiate implicit calls. These methods are defined by Android's specification. As new sub-call graphs are linked and connected, they are traversed in a similar fashion. By doing so, we are 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.

Finally, the third step in deriving the *BM* is to remove all of the non-root nodes from the call-graph of an app and connecting the remaining nodes. The *BM* for the Banking app is depicted in Figure 12.4b. The *BM* only captures a high level behavior of the app in terms of the event interactions and does not include unnecessary details about the sequences of explicit method calls within the Android components.

It must be noted that the obtained call graph and hence the *BM* only represent the

Algorithm 3: Implicit Call Extraction

Input: CG : set of sub-call graphs, ψ : set of implicit callers
Output: Υ : implicit calls

```
1 foreach c ∈ CG do
2     rootNodes ← c.getRoot()
3 foreach c ∈ CG do
4     lNodes ← c.getLeafNodes()
5     foreach l ∈ lNodes do
6         if l ∈ ψ then
7             d ← l.getDestinationNode()
8             if d ∈ rootNodes then
9                 Υ.Add(l, d)
```

chain of possible method calls regardless of constraints, i.e., the call graph of an app does not include any information about conditions and the control flow of an app. As will be explained in detail later, the *BM* is only used to generate the *Drivers* for symbolic execution and not directly to generate the sequences in test cases. In other words, it is used to navigate within the app to determine all the ways in which it receives user inputs, system notifications, starts/stops/resumes activities and services, interacts using Intents, etc.

12.5.2 Interface Model

The *IM* provides information about all of the input interfaces of an app, such as the widgets and input fields belonging to an Activity. It also includes information about the application-level and system-level Intent events handled by each Activity. The *IM* is obtained by combining and correlating the information contained in the configuration files and meta-data included in Android APK (such as Android Manifest and layout XML files). For each activity the corresponding layout file (recall Figure 12.2c) is parsed to obtain all information on each widget such as name, id, input type and so on. As described in the next section, the information provided by the *IM* is used to identify possible symbolic values in the program.

12.6 Symbolic Execution for Android

Symbolically executing applications in Android face three major challenges because Android apps are (1) event-driven, (2) prone to path-divergence, and (3) compiled into Dalvik bytecode. In this section, I explain how SIG-Droid's symbolic execution engine addresses these challenges.

12.6.1 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 [32], 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, I use the *BM* 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 *BM* in Figure 12.4b, if we start at `MainActivity.onCreate()` and follow through with `TransferAtivity.onCreate()` and `Transfer.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 12.1 illustrates a sample Driver for banking app generated in this way using the sample *BM* of Figure 12.4b. It contains two sequence of events, i.e., creates a `TransferActivity` 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 Transfer button by calling

`onClick` method.

Note that since the *BM* 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 12.7, the Drivers are not used 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.

12.6.2 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 [38]. 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

Listing 12.1: Sample Driver for banking app.

```
public static void main(String[] args) {
    try {
        View v = new View(null);
        TransferActivity ta = new TransferActivity();
        Transfer t = ta.new Transfer();
        ta.onCreate();
        t.onClick(v);
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}
```

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) [16] 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 , we need to first transform the app into the corresponding Java byte-code representation.

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. Stubs that return random values within a reasonable range are used, 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 we need to provide stub classes to resolve the byte-code incompatibility with JVM, but we 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 Chapter 2 that in Android inter-component messaging is achieved through Intents). In the case of banking app, as

shown in Figure 12.2b, `TransferActivity` uses the `startActivity` method of the Android library class `Activity.java` to start the app's `AccountActivity` that displays the accounts and their respective balances. It creates an Intent in which the source and destination activities along with the values to be carried are specified. In this case, the appropriate logic is provided 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, a mock for the `Intent.java` is created to address the path-divergence problem in cases where the payload is a symbolic value. As shown in Figure 12.2b, 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, a custom implementation of `putExtra` and `getExtra` methods are provided 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, a custom implementation of a hash map that holds primitive values is used. Consequently, in SIG-Droid, the implementation of the `putExtra` and `getExtra` methods, the customized hash map implementation is used 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 transfer amount in the banking 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.

12.7 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 *BM* may contain several constraints, thus it may result in multiple execution paths. For instance, if `amountValue` is a symbolic value in `TransferActivity` in Figure 12.2, the Driver in Listing 12.1 would result in two different execution paths: One where the `amountValue` is less than $5000, and another where it is greater. Hence, the report for each Driver may result in several tests.

Moreover, as mentioned in Section 12.5, the call graph of 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 *BM* may be invalid sequences of events, meaning that the constraints may prevent the execution of certain events. To ensure that only valid sequences of events are generated 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 12.2 shows the symbolic execution report for the Driver of Listing 12.1.

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 the *IM* is used to determine the event generator corresponding to each event handler in the report. For example, `Transfer.onClick` handler method in Listing 12.2 is the handler for the `Transfer` button on `TransferActivity` screen of Figure 12.2.

Listing 12.3 illustrates one of the Robotium test cases generated by SIG-Droid that corresponds to the report shown in Listing 12.2. 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

```
<?xml version="1.0" encoding="utf-8"?>
<Report>
  <Path id="1"
    <Activity name="TransfersActivity">
      <MethodCall name="Transfer.onClick()">
        <SoldvedVariable name="amountValue" value="1" />
      </MethodCall>
    </Activity>
  </Path>
  <Path id="2"
    <Activity name="TransfersActivity"
      <MethodCall name="Transfer.onClick()">
        <SoldvedVariable name="amountValue" value="5001" />
      </MethodCall>
    </Activity>
  </Path>
</Report>
```

Listing 12.2: Symbolic Execution report for Driver in Listing 12.1.

Listing 12.3: Code snippet of a Robotium test automatically generated by SIG-Droid for TransferActivity.

```
public class TransferActivityTest_1 extends
      ActivityInstrumentationTestCase2<TransferActivity> {

    private Solo solo;
       ...
       @smoke
       public void testMethod() throws Exception {
          solo.enterText(0,'5001');
          solo.clickOnButton('Transfer');
    }
    ...
}
```

on that activity, and then clicks on `Transfer` button.

12.8 Limitations

One of the limitations of SIG-Droid is the *Driver* generation. Currently the sequence of events is generated through a depth first search on the *BM*, 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.

Chapter 13: Conclusion

In this dissertation, I provided a detailed description of EvoDroid, an approach for automated system testing of Android applications. I now conclude my dissertation by enumerating the contributions of EvoDroid, threats to its validity and avenues for future work.

In summary, evolutionary testing is typically applied at the unit level, as there is no effective approach to pass on the genetic make-up of these individuals to the next generations at the system/app level. The crossover strategy does not consider which input and event genes are coupled to which part of the app, hence, it is not able to preserve the genetic makeup of parents in any meaningful way. It mixes genetic information from different parts of the app along an execution path or along different execution paths. As a result, it produces tests that are likely to be either not executable or inferior to both parents. In evolutionary search, the inability to promote and pass on the genetic makeup of good individuals to the next generations is highly detrimental to its effectiveness.

EvoDroid is the first evolutionary testing framework targeted at Android. The most notable contribution of EvoDroid is its ability to overcome the common shortcoming of using evolutionary techniques for system testing. EvoDroid overcomes the crossover issue by leveraging the knowledge of how Android specifies and constrains the way apps can be built. It analyzes apps and infers models of their interface and behavior. Using these models EvoDroid generates test cases reaching deep into the code in segments, i.e., sections of code that are amenable to evolutionary testing without the possibility of generating invalid test cases. Since a key concern in search-based testing is the execution time of the algorithm, EvoDroid runs the test cases in parallel on multiple Android emulators, thus achieving several orders of magnitude improvement in execution time.

Using the prototype implementation and evaluation details provided in Chapter 10, I have validated the approach and its properties.

13.1 Contributions

The following is a concrete list of contributions of this research:

- **Automated Android testing framework**: Fully automated framework to perform system-wide testing of event driven applications built on ADFs such as Android. The cloud testing framework serves as the basis for potential commercialization for automated Testing as a Service.

- **Fundamental contribution to evolutionary testing**: A unique evolutionary testing algorithm, called segmented evolutionary testing, that eliminates the high level representation and crossover issues faced by existing evolutionary testing approaches.

- **Method for generating connected call graphs in event based systems**: Automated program analysis techniques to extract the models of the applications, and connect the call-graph model of event-driven applications to form the entire search graph. By analyzing the specifications of the underlying framework, EvoDroid connects the disconnected call graphs in event driven frameworks such as Android.

- **A combination of evolutionary testing and GUI crawling**: An approach for dynamically extracting the application models necessary to support automated system testing. EvoDroid combined existing GUI crawling techniques with segmented evolutionary testing to extract the models without requiring the source code.

- **Foundation framework for targeted testing**: A foundation to serve use cases requiring targeted program execution as presented in Chapter 11. EvoDroid provided a mechanism to mitigate the general reachability problem in software testing.

13.2 Threats to Validity

Although my approach has fundamentally contributed to evolutionary testing and shown to be significantly better than existing tools and techniques for Android apps, in the worst case scenario it can degrade due to its inability to systematically reason about input conditions. This is a known limitation of search based algorithms, such as evolutionary testing. To mitigate this issue, I have worked on SIG-Droid, the symbolic input generation tool described in Chapter 12. SIG-Droid extends Java Pathfinder, which symbolically executes pure Java code and solves input constraints, to work on Android. The good news, however, is that based on my empirical study on the complexity of Android apps from Chapter 9, on average Android apps are event heavy and not very input constrained.

Another weakness of EvoDroid is not being able to automatically generate accurate models for apps. There is a significant variability in the way the app code is generally written as well as Android version fragmentation, code generation via reflection, and anonymous class declarations among other reasons. As a result, the models may in fact be incomplete. However, I have mitigated this issue by providing a static and dynamic approach as described in Chapter 6 and Chapter 7. Additionally, as mentioned earlier and evaluated in Chapter 10, EvoDroid is able to work on partial and/or incomplete models.

13.3 Future Work

I plan to integrate SIG-Droid into EvoDroid and use both techniques in tandem to complement one another. As shown in Figure 13.1, EvoDroid would find the drivers or the correct sequence of events necessary in SIG-Droid and then SIG-Droid would be applied to solve the input constraints locally in each segment.

As alluded to earlier, symbolic-based test generation provides a systematic approach to reason about the input values, but it is no better than random search in finding the right sequence of user actions or system events. On the contrary, evolutionary-based test generation is effective in guiding the search to find the right sequence of actions and events,

but it is no better than random search in finding the right input values.

I plan use symbolic execution within the evolutionary process of generating populations, such that within each evolutionary cycle, instead of randomly selecting the input values, we actually use the values resulting from targeted symbolic execution of the program. This allows us to bear the benefits of each technique and likely to result in much faster and significantly less number of iterations to automatically generate system level tests.

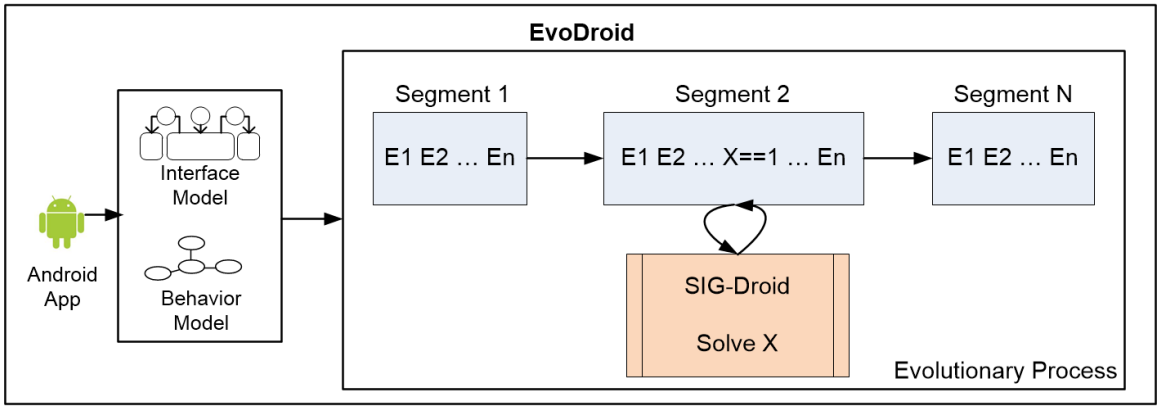


Figure 13.1: Segmented Evolutionary Testing With Constraint Solver.

Bibliography

Bibliography

[1] Activity manager, http://developer.android.com/reference/android/app/ activity-manager.html.

[2] Amazon Web Services CLI, http://aws.amazon.com/cli/.

[3] Amazon Web Services, http://aws.amazon.com/.

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

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

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

[7] Android emulator, http://developer.android.com/tools/help/emulator.html.

[8] Android intent fuzzer, https://www.isecpartners.com/mobile-security-tools/intent-fuzzer.html.

[9] Android logcat, http://developer.android.com/tools/help/logcat.html.

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

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

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

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

[14] Calabash, http://www.moncefbelyamani.com/ios-automated-testing-with-calabash-cucumber-ruby.

[15] Coverity, http://www.coverity.com.

[16] Dalvik, http://code.google.com/p/dalvik/.

[17] Ecj, http://cs.gmu.edu/ eclab/projects/ecj/.

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

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

[20] Findbugs, http://findbugs.sourceforge.net.

[21] Hierarchy viewer, http://developer.android.com/tools/help/hierarchy-viewer.html.

[22] Java pathfinder, http://babelfish.arc.nasa.gov/trac/jpf/.

[23] Jython, http://www.jython.org/.

[24] Klockwork, http://www.clockwork.com.

[25] MoDisco, http://www.eclipse.org/modisco/.

[26] Monkey runner, http://developer.android.com/tools/help/monkeyrunner_concepts.html.

[27] Monkeytalk, http://www.gorillalogic.com/monkeytalk.

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

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

[30] Sourceforge eclipse metrics.

[31] Subversion, https://subversion.apache.org/.

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

[33] Alsmadi, I., Alkhateeb, F., Maghayreh, E., Samarah, S., and Doush, I. A. Effective generation of test cases using genetic algorithms and optimization theory. *Journal of Communication and Computer 7*, 11 (2010), 72–82.

[34] Amalfitano, D., Fasolino, A., and Tramontana, P. 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.

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

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

[37] Ammann, P., and Offutt, J. *Introduction to Software Testing*. Cambridge University Press, Jan. 2008.

[38] Anand, S. *Techniques to Facilitate Symbolic Execution of Real-world Programs*. Ph.D., Georgia Institute of Technology, 2012.

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

[40] Andersson, M. *Software Security Testing : A Flexible Architecture for Security Testing*. 2008.

[41] Avgerinos, T., Cha, S. K., Rebert, A., Schwartz, E. J., Woo, M., and Brumley, D. Automatic Exploit Generation. *Commun. ACM 57*, 2 (Feb. 2014), 74–84.

[42] Azim, T., and Neamtiu, I. 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* (2013), OOPSLA '13, pp. 641–660.

[43] Back, T. *Evolutionary algorithms in theory and practice*. Oxford Univ. Press, 1996.

[44] Baresi, L., Lanzi, P. L., and Miraz, M. Testful: An evolutionary test approach for java. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation* (2010), ICST '10, pp. 185–194.

[45] Bauersfeld, S., Wappler, S., and Wegener, J. A metaheuristic approach to test sequence generation for applications with a gui. In *Proceedings of the Third International Conference on Search Based Software Engineering* (2011), SSBSE'11, pp. 173–187.

[46] Berndt, D., and Watkins, A. Investigating the performance of genetic algorithm-based software test case generation. In *Eighth IEEE International Symposium on High Assurance Systems Engineering, 2004. Proceedings* (Mar. 2004), pp. 261–262.

[47] Bickford, J., Lagar-Cavilla, H. A., Varshavsky, A., Ganapathy, V., and Iftode, L. Security Versus Energy Tradeoffs in Host-based Mobile Malware Detection. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2011), MobiSys '11, ACM, pp. 225–238.

[48] Bird, D., and Munoz, C. Automatic generation of random self-checking test cases. *IBM Systems Journal 22*, 3 (1983), 229–245.

[49] Bo, J., Xiang, L., and Xiaopeng, G. MobileTest: A Tool Supporting Automatic Black Box Test for Software on Smart Mobile Devices. In *Proceedings of the Second International Workshop on Automation of Software Test* (Washington, DC, USA, 2007), AST '07, IEEE Computer Society, pp. 8–.

[50] Burguera, I., Zurutuza, U., and Nadjm-Tehrani, S. Crowdroid: Behavior-based Malware Detection System for Android. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices* (New York, NY, USA, 2011), SPSM '11, ACM, pp. 15–26.

[51] Cadar, C., Dunbar, D., and Engler, D. 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* (Berkeley, CA, USA, 2008), OSDI'08, USENIX Association, pp. 209–224.

[52] Candea, G., Bucur, S., and Zamfir, C. Automated software testing as a service. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (2010), SoCC '10, pp. 155–160.

[53] Cheng, J., Wong, S. H., Yang, H., and Lu, S. SmartSiren: Virus Detection and Alert for Smartphones. In *Proceedings of the 5th International Conference on Mobile Systems, Applications and Services* (New York, NY, USA, 2007), MobiSys '07, ACM, pp. 258–271.

[54] Chin, E., Felt, A. P., Greenwood, K., and Wagner, D. Analyzing inter-application communication in android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services* (2011), MobiSys '11, pp. 239–252.

[55] Choi, W., Necula, G., and Sen, K. 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* (2013), OOPSLA '13, pp. 623–640.

[56] Ciortea, L., Zamfir, C., Bucur, S., Chipounov, V., and Candea, G. Cloud9: A software testing service. *SIGOPS Oper. Syst. Rev. 43*, 4 (Jan. 2010), 5–10.

[57] Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM Trans. Comput. Syst. 32*, 2 (June 2014), 5:1–5:29.

[58] Enck, W., Octeau, D., McDaniel, P., and Chaudhuri, S. A study of android application security. In *Proceedings of the 20th USENIX Conference on Security* (2011), SEC'11, pp. 21–21.

[59] Enck, W., Ongtang, M., and McDaniel, P. On Lightweight Mobile Phone Application Certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2009), CCS '09, ACM, pp. 235–245.

[60] Felt, A. P., Hanna, S., Chin, E., Wang, H. J., and Moshchuk, E. Permission re-delegation: Attacks and defenses. In *In 20th Usenix Security Symposium* (2011).

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

[62] Forrester, J. E., and Miller, B. P. An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. In *Proceedings of the 4th Conference on USENIX Windows Systems Symposium - Volume 4* (Berkeley, CA, USA, 2000), WSS'00, USENIX Association, pp. 6–6.

[63] Fuchs, A. P., Chaudhuri, A., and Foster, J. S. SCanDroid: Automated Security Certification of Android Applications. Tech. Rep. CS-TR-4991, Department of Computer Science, University of Maryland, College Park, November 2009.

[64] Ganov, S., Killmar, C., Khurshid, S., and Perry, D. E. 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* (2009), ICFEM '09, pp. 69–87.

[65] Gibler, C., Crussell, J., Erickson, J., and Chen, H. AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. In *Trust and Trustworthy Computing*, S. Katzenbeisser, E. Weippl, L. J. Camp, M. Volkamer, M. Reiter, and X. Zhang, Eds., no. 7344 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 291–307.

[66] Godefroid, P., Kiezun, A., and Levin, M. Grammar-based whitebox fuzzing. In *ACM SIGPLAN Notices* (2008), vol. 43, pp. 206–215.

[67] Godefroid, P., Klarlund, N., and Sen, K. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2005), PLDI '05, ACM, pp. 213–223.

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

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

[70] Grace, M., Zhou, Y., Zhang, Q., Zou, S., and Jiang, X. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2012), MobiSys '12, ACM, pp. 281–294.

[71] Gross, F., Fraser, G., and Zeller, A. Exsyst: Search-based gui testing. In *Software Engineering (ICSE), 2012 34th International Conference on* (June 2012), pp. 1423–1426.

[72] Gross, F., Fraser, G., and Zeller, A. Search-based system testing: High coverage, no false alarms. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis* (2012), ISSTA 2012, pp. 67–77.

[73] Hackner, D. R., and Memon, A. M. Test Case Generator for GUITAR. In *Companion of the 30th International Conference on Software Engineering* (New York, NY, USA, 2008), ICSE Companion '08, ACM, pp. 959–960.

[74] Henderson-Sellers, B. *Object-Oriented Metrics, measures of complexity.* Prentice Hall, 1996.

[75] Hovemeyer, D., and Pugh, W. Finding Bugs is Easy. *SIGPLAN Not. 39*, 12 (Dec. 2004), 92–106.

[76] Hu, C., and Neamtiu, I. Automating gui testing for android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test* (2011), AST '11, pp. 77–83.

[77] Inkumsah, K., and Xie, T. 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* (2008), ASE '08, pp. 297–306.

[78] Isohara, T., Takemori, K., and Kubota, A. Kernel-based Behavior Analysis for Android Malware Detection. In *2011 Seventh International Conference on Computational Intelligence and Security (CIS)* (Dec. 2011), pp. 1011–1015.

[79] Jensen, C. S., Prasad, M. R., and Møller, A. Automated testing with targeted event sequence generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis* (2013), ISSTA 2013, pp. 67–77.

[80] Karami, M., Elsabagh, M., Najafiborazjani, P., and Stavrou, A. Behavioral Analysis of Android Applications Using Automated Instrumentation. In *2013 IEEE 7th International Conference on Software Security and Reliability-Companion (SERE-C)* (June 2013), pp. 182–187.

[81] Kim, C. H. P., Batory, D. S., and Khurshid, S. Reducing combinatorics in testing product lines. In *Proceedings of the Tenth International Conference on Aspect-oriented Software Development* (New York, NY, USA, 2011), AOSD '11, ACM, pp. 57–68.

[82] Kim, J.-M., and Kim, J.-S. Androbench: Benchmarking the storage performance of android-basex mobile devices. In *Frontiers in Computer Education*. Springer, 2012, pp. 667–674.

[83] King, J. C. A new approach to program testing. In *Proceedings of the International Conference on Reliable Software* (1975), pp. 228–233.

[84] Landi, W. Undecidability of Static Analysis. *ACM Lett. Program. Lang. Syst. 1*, 4 (Dec. 1992), 323–337.

[85] Li, D., Hao, S., Gui, J., and Halfond, W. G. An empirical study of the energy consumption of android applications. In *The Intl. Conf. on Software Maintenance and Evolution* (2014).

[86] Lin, C.-M., Lin, J.-H., Dow, C.-R., and Wen, C.-M. Benchmark dalvik and native code for android system. In *Innovations in Bio-inspired Computing and Applications (IBICA), 2011 Second International Conference on* (2011), IEEE, pp. 320–323.

[87] Machiry, A., Tahiliani, R., and Naik, M. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (2013), ESEC/FSE 2013, pp. 224–234.

[88] Mahmood, R., Esfahani, N., Kacem, T., Mirzaei, N., Malek, S., and Stavrou, A. A whitebox approach for automated security testing of android applications on the cloud. In *2012 7th International Workshop on Automation of Software Test (AST)* (June 2012), pp. 22 –28.

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

[90] Mariani, L., Pezze, M., Riganelli, O., and Santoro, M. Autoblacktest: Automatic black-box testing of interactive applications. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on* (April 2012), pp. 81–90.

[91] Martin, R. Oo design quality metrics : an analysis of dependencies. *ROAD 1995 2*, 3 (1995).

[92] McCabe, T. J. A complexity measure. *Software Engineering, IEEE Transactions on*, 4 (1976), 308–320.

[93] Memon, A., Banerjee, I., and Nagarajan, A. Gui ripping: Reverse engineering of graphical user interfaces for testing. In *Proceedings of the 10th Working Conference on Reverse Engineering* (2003), WCRE '03, pp. 260–.

[94] Memon, A. M., Pollack, M. E., and Soffa, M. L. Automated test oracles for guis. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering: Twenty-first Century Applications* (2000), SIGSOFT '00/FSE-8, pp. 30–39.

[95] Miller, C., and Mulliner, C. Fuzzing the Phone in your Phone. In *Black Hat Technical Security Conference USA* (2009).

[96] Mirzaei, N., Malek, S., Păsăreanu, C. S., Esfahani, N., and Mahmood, R. Testing android apps through symbolic execution. *SIGSOFT Softw. Eng. Notes 37*, 6 (Nov. 2012), 1–5.

[97] Nguyen, B. N., Robbins, B., Banerjee, I., and Memon, A. Guitar: an innovative tool for automated testing of gui-driven software. *Automated Software Engineering 21*, 1 (2014), 65–105.

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

[99] Oehlert, P. Violating assumptions with fuzzing. *IEEE Security Privacy 3*, 2 (Mar. 2005), 58–62.

[100] Ongtang, M., McLaughlin, S., Enck, W., and McDaniel, P. Semantically rich application-centric security in Android. *Security and Communication Networks 5*, 6 (June 2012), 658–673.

[101] Ramalingam, G. The Undecidability of Aliasing. *ACM Trans. Program. Lang. Syst. 16*, 5 (Sept. 1994), 1467–1471.

[102] Rauf, A., Jaffar, A., and Shahid, A. A. Fully automated gui testing and coverage analysis using genetic algorithms. *International Journal of Innovative Computing, Information and Control (IJICIC) Vol 7* (2011).

[103] Sen, K., Marinov, D., and Agha, G. 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* (2005), ESEC/FSE-13, pp. 263–272.

[104] Shabtai, A., Fledel, Y., Kanonov, U., Elovici, Y., and Dolev, S. Google Android: A State-of-the-Art Review of Security Mechanisms. *arXiv:0912.5101 [cs]* (Dec. 2009). arXiv: 0912.5101.

[105] Shabtai, A., Kanonov, U., Elovici, Y., Glezer, C., and Weiss, Y. Andromaly: a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems 38*, 1 (Jan. 2011), 161–190.

[106] Shahamiri, S. R., Kadir, W. M. N. W., and Mohd-Hashim, S. Z. A comparative study on automated software test oracle methods. In *Proceedings of the 2009 Fourth International Conference on Software Engineering Advances* (2009), ICSEA '09, pp. 140–145.

[107] Sutton, M., Greene, A., and Amini, P. *Fuzzing: Brute Force Vulnerability Discovery*. Pearson Education, June 2007.

[108] Takanen, A., DeMott, J. D., and Miller, C. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, Jan. 2008.

[109] Tonella, P. Evolutionary testing of classes. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis* (2004), ISSTA '04, pp. 119–128.

[110] Wappler, S., and Lammermann, F. Using evolutionary algorithms for the unit testing of object-oriented software. In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation* (2005), GECCO '05, pp. 1053–1060.

[111] Wei, T.-E., Mao, C.-H., Jeng, A., Lee, H.-M., Wang, H.-T., and Wu, D.-J. Android Malware Detection via a Latent Network Behavior Analysis. In *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)* (June 2012), pp. 1251–1258.

[112] White, L., and Almezen, H. 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.

[113] Yang, W., Prasad, M. R., and Xie, T. 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* (2013), FASE'13, pp. 250–265.

[114] Yuan, X., Cohen, M., and Memon, A. M. 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* (New York, NY, USA, 2007), ASE '07, ACM, pp. 405–408.

[115] Yuan, X., and Memon, A. M. Generating event sequence-based test cases using gui runtime state feedback. *IEEE Trans. Softw. Eng. 36*, 1 (Jan. 2010), 81–95.

[116] Zhifang, L., Bin, L., and Xiaopeng, G. Test Automation on Mobile Device. In *Proceedings of the 5th Workshop on Automation of Software Test* (New York, NY, USA, 2010), AST '10, ACM, pp. 1–7.

BIOGRAPHY

Riyadh Mahmood started his PhD with the Department of Computer Science at George Mason University (GMU) in 2007. His current research mainly focuses on Android testing, software engineering, mobile/distributed software systems, and autonomic computing. Riyadh received his MS degree in Information Technology from Virginia Tech in 2004 and his BS degree in Computer Engineering from Virginia Tech in 2000.