Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning

Wonae Choi

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2018-149
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-149.html

December 1, 2018

Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning

by

Philip Wontae Choi

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Koushik Sen, Chair
George Necula
Björn Hartmann
Sara McMains

Fall 2017

Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning

Abstract

Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning

by

Philip Wontae Choi

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Koushik Sen, Chair

In recent years, there has been a significant surge in the use and development of apps for smartphones and tablets. The complexity of mobile apps mostly lies in their graphical user interfaces (GUIs) since the computational part of these apps are either simple or is delegated to a backend server. Therefore, testing efforts of these apps mostly focus on the behavior of graphical user interfaces.

In this thesis, we address two problems in GUI testing, test suite generation and test suite reduction, in the context of Android apps. In the first part of the thesis, we propose the SwiftHand algorithm for automatically generating sequences of test inputs for GUI apps. The algorithm uses machine learning to learn a model of the app during testing, uses the learned model to generate user inputs that visit unexplored states of the app, and uses the execution of the app on the generated inputs to refine the model. A key feature of the algorithm is that it avoids restarting the app, which is a significantly more expensive operation than executing the app on a sequence of inputs. We evaluate SwiftHand on several Android apps, showing that it can outperform a simple fuzz-testing algorithm and a traditional active-learning based testing algorithm.

In the second part of the thesis, we address the problem of reducing GUI test suites generated by automated testing tools. Automated testing tools, including SwiftHand, have been shown to be effective in achieving good test coverage and in finding bugs. Being automated, these tools typically run for a long period, e.g. for several hours, until the time budget for testing expires. However, these tools are not good at generating concise regression test suites that could be used for testing in incremental development of the apps and in regression testing. We propose DetReduce, a heuristic technique that helps to create a small regression test suite for a GUI app given a large test suite generated by running an automated GUI testing tool. The key insight is that if we can identify and remove some common forms of redundancies introduced by existing automated GUI testing tools, then we can drastically lower the time required to minimize a GUI test suite. We evaluate DetReduce on several Android apps, showing that it can reduce the size and running time of an automatically generated test suite by at least an order of magnitude.

To my wife, to my parents, and to my brother.

Contents

Acknowledgments

First, I would like to thank my advisors, Koushik Sen and George Necula. They have been great mentors and teachers for past few years. I am grateful for their patient mentoring and tireless effort to help my study and research, especially while I was passing through difficult times during my Ph.D program. I would also like to thank Björn Hartmann and Sara McMains for serving in my Quals and dissertation committee.

I would like to thank Wonchan Lee, Patrick Li, and David Wagner for providing important feedback on my research. The discussions I had with these three people at the early stage of this work helped to shape some fundamental ideas.

I thank to my collaborators and coauthors: Domagoj Babic, Satish Chandra, Liang Gong, Colin S. Gordon, Benjamin Mehne, Manu Sridharan, Jayanthkumar Kannan, Frank Tip, and Wenyu Wang. I learned a lot by working with all of them. Special thanks to Satish Chandra for supporting the SJS project and hosting me at Samsung Research America. I also thank to Jayanthkumar Kannan, my mentor at Google, for an enjoyable internship.

I am grateful to all my friends I have worked at Berkeley, including Benjamin Brock, Sarah Chasin, Kevin Chen, Peter Xi Chen, Ankush Desai, Rafael Tupynamba Dutra, Tayfun Elmas, Joel Galenson, Cindy Rubio Gonzáleze, Sangjin Han, Hokeun Kim, Woojong Koh, Ali Sinan Köksal, Donggyu Lee, Yunsup Lee, Caroline Lemieux, Phitchaya Mangpo Phothilmathana, Cuong Nguyen, Rohan Padhye, Chang-Seo Park, Michael Pradel, Evan Pu, Xeuhai Qian, Philip Reames, Alex Reinking, Neeraja Yadwadkar, Christina Teodoropol, and Nishant Totla. I will miss research discussions, critical feedback on my papers, and all the small talks we had at Soda hall. Special thanks to Roxana Infante and Audrey Sillers for their organizational help.

I would like to take this opportunity to thank to those who helped and led me to PL/SE research field, including Seungjae Lee, Jaeseung Ha, Kyunglyul Hyun, Soonho Kong, Daejun Park, and Kwangkeun Yi. I owe specially thanks to Kwangkeun Yi, my adviser during my Master's program at Seoul National University, for all his valuable advice and continuous support.

Finally and foremost, I would like to thank my wife Hyewon Kim for constant support, understanding, and love. I thank my parents and brother. I am deeply grateful for all their love and support.

 This work supported by NSF Grants CCF- 1017810, CCF-0747390, CCF-1018729, CCF-1423645, CCF-1409872, and CCF-1018730, and a gift from Samsung.

Chapter 1

Introduction

Recent years have seen a significant surge in the use and development of smartphone apps. A developer survey performed by StackOverflow [2] suggests that 23% of developers are now involved in smartphone app development, and AppBrain [1] reports that more than 800,000 new apps appeared in the Google Play Store in 2016 alone. These numbers demonstrate that apps are an integral part of the digital ecosystem.

The complexity of such mobile apps lies primarily in their graphical user interfaces (GUIs) since the computational components of these apps are either relatively simple or are delegated to a backend server. A similar situation exists for web and desktop apps based on software-as-a-service architecture, for which the client-side components consist largely of user interface code. As such, testing efforts on these apps focus on the behavior of graphical user interfaces.

1.1 GUI Testing in Practice

This thesis focuses on user interface testing for Android apps. We believe that our techniques are also applicable to other mobile and browser platforms. In practice, Android developers use test scripting tools [37, 38, 95] or *Monkey* [36], a random automatic user input generation tool, to test their apps [62].

- Test scripting tools for Android apps, such as *Espresso* [38] or *Robotium* [95], require developers to manually script sequences of user inputs in a programming language. According to a recent survey by Kochhar et al. [62], most smartphone app developers use test scripts to test their apps. In this approach, a developer writes small programs (test scripts) that inject inputs to the target app and check assertions during the execution of the app. Once implemented, a test script can be used as a push-button solution generating results that are easy to understand. However, manual scripting is labor-intensive and error-prone. Furthermore, the GUI of an app tends to change along with the evolution of the app, and manually written test scripts must therefore be updated whenever the GUI of the app has been modified. Record-and-replay tools [34, 38, 53] avoid directly writing test scripts, but nonetheless free developers from neither

test case creation nor maintenance—developers must still manually test apps once and update test scripts whenever there is an update to the app.

- Developers often use *Monkey* to complement test scripts, focusing their efforts on writing test scripts to check important invariants of their apps and employing *Monkey* to find shallow bugs that are mistakenly introduced. The strength of *Monkey* is its robustness (it runs for every app on every environment), which stems from its simplicity. A typical use of the *Monkey* tool involves generating clicks at random positions on the screen without any consideration for the actual control interface, for which elements have already been clicked, or for the sequence of steps to arrive at the current configuration. However, it is not surprising that such testing has difficulty exploring enough of the user interface, especially parts that can be reached only after a specific sequence of inputs. Choudhary et al.'s survey [24] suggests that *Monkey* does achieve good test coverage by injecting user actions quickly, despite this limitation. However, it is also known that *Monkey*'s strategy to rapidly inject many user actions simultaneously makes it difficult to replay test cases [107]. More specifically, *Monkey* often fails to inject a part of planned events, but still reports those events as successfully injected. Therefore, even if *Monkey* finds a bug, it can be difficult to reproduce the bug or to identify the specific sequence of actions that generated it depending on the information provided by *Monkey*.

1.2 Our Goals

In this research, we address the following two problems in GUI testing.

- First, we address the problem of automatically generating test cases for a GUI app without requiring a priori knowledge about the app. To this end, we have developed an automated GUI testing algorithm that aims to maximize the test coverage within a limited time period. There are two requirements for an automated GUI test generation tool: a) the tool should achieve better test coverage than randomized algorithms, and b) it should allow recording the sequences of inputs used by the testing algorithm to reuse later as a regression test suite for the app, or to understand the bugs detected while running the automatically generated test cases.

- Second, we address the problem of constructing a small GUI regression test suite by reducing automatically generated GUI test cases. Automated GUI test generation tools typically produce test suites containing thousands of test cases. Each test case potentially contains tens to thousands of user actions. Such a large number of test cases cannot be used for regression testing because it would then take several hours to run; regression tests must run faster so that they can be used frequently during development. The technical question is to generate such small test suites by reducing large, automatically generated test suites. Note that even GUI test cases [45] and GUI

test suites [74] created by human experts are reported to be reducible. Therefore, it is reasonable to expect that test suites generated by a machine are also reducible.

Solving these two problems is important for allowing developers both to jump-start GUI testing and to generate test suites. When beginning to test an app, a developer can focus on writing test cases that check important invariants of the app and simultaneously run an automated testing tool to generate an efficient test suite (instead of Monkey). The developer can subsequently run a GUI test suite reduction tool to shrink the size of the machine-generated test suite. Once a small test suite is obtained this way, the developer can use it as a regression test suite after inserting assertions into the test suite or adding invariant checks to the app.

1.3 Contributions and Outline

In this dissertation, we propose two novel algorithms to address problems in GUI testing.

Automated GUI Testing with Minimal Restart and Approximate Learning

We propose SwiftHand, an algorithm for testing GUI apps automatically, without human intervention. The algorithm generates sequences of test inputs for Android apps for which we do not have an existing model of the GUI. The goal is to *achieve code coverage quickly* by learning and exploring an abstraction of the model of the app's GUI. One insight behind this technique is that the automatic construction of a model of the user interface and the testing of this interface are tasks that can cooperate in a mutually beneficial way. The SwiftHand algorithm has two key features. First, the algorithm avoids restarting the app, which is a significantly more expensive operation than executing the app on a sequence of inputs. Second, SwiftHand learns an approximate model rather than a precise model. An important insight behind our testing algorithm is that if our goal is simply guiding the test executions into unexplored elements of the state space, this does not require the computationally intensive work of learning a precise model of the app. Our experimental results show that SwiftHand can achieve significantly better coverage than either traditional random testing or learning-based testing in a given time budget. Our algorithm also reaches peak coverage more quickly than both random and learning-based testing.

Minimizing GUI Test Suites for Regression Testing

We also propose DetReduce, a GUI test suite reduction algorithm that can scalably and effectively minimize large test suites. The key insight behind our technique is that if we can identify and remove common redundancies introduced by existing automated GUI test generation tools, then we can significantly lower the time required to minimize a GUI test suite.

After manually analyzing several sources of redundancies in the test suites generated by the automated GUI test generation tool, SwiftHand, we identified three kinds of redundancy that are common in these test suites: 1) some test cases can be safely removed from a test suite without affecting the code and screen coverage, 2) within a test case, certain loops can be eliminated without decreasing coverage, and 3) many test cases share common subsequences of actions, the repeated execution of which can be avoided by combining fragments from different action sequences. Based on these observations, we have developed an algorithm that removes such redundancies one-by-one while ensuring that this does not reduce the overall code and screen coverage of the resulting test suite. We applied DetReduce to test suites generated from SwiftHand and a randomized test generation algorithm, and found that on average DetReduce could reduce a test suite by a factor of 16.9× in size and 14.7× in running time. We also observed that the test suites reduced by DetReduce retain all distinct exceptions raised while executing the original test suites.

The rest of this thesis is organized as follows. Chapter 2 describes the SwiftHand algorithm. We first introduce SwiftHand using examples, then provide a formal definition of the algorithm. Chapter 3 describes the DetReduce algorithm. We detail three types of common redundancies in automatically generated test suites, review the current high-level ideas to detect and remove such redundancies, and then formally describe the DetReduce algorithm. In Chapter 4, we describe the implementation details of our Android testing infrastructure, on which SwiftHand and DetReduce are built. Chapter 5 summarizes the evaluation of SwiftHand and DetReduce on several real-world Android apps. In Chapter 6, we discuss related work. Chapter 7 concludes the thesis with a discussion of the limitations of our work and the possible future research.

Chapter 2

SwiftHand, a GUI Testing Algorithm with Minimal Restart and Approximate Learning

In this chapter, we consider the problem of automatically generating sequences of test inputs for Android apps for which we do not have an existing model of the GUI. The goal is to *achieve code coverage quickly*. One promising approach is to use a learning-based testing algorithm [11, 98]. A learning-based algorithm is capable of gradually learning a behavioral model of the target app throughout the testing and to guide testing using this model. However, our experience shows that previous learning-based testing techniques ignore the execution cost of restarting an app. All automatic exploration algorithms will occasionally need to restart the target app, in order to explore additional states reachable only from the initial state. Such a restart operation is also required to clear all persistent data of the app. For an Android app, a restart operation can be implemented by first closing the app, removing all app-specific persistent data, and finally starting the app again. Our experiments show that this particular restart operation takes 30 seconds on an Android emulator, which is significantly longer than the time required to explore any other transition, such as a user input. Currently, our implementation waits for up to 5 seconds after sending a user input, to ensure that all handlers have finished. Since the cost of exploring a transition to the initial state (a restart) is an order of magnitude more than the cost of any other transition, an efficient exploration and learning algorithm will minimize the number of restarts.

In this chapter, we propose a new learning-based testing algorithm based on two key observations:

1. It is possible to reduce the use of app restarts, as most user interface screens of an Android app can be reached from other screens by triggering a sequence of user inputs, such as "back" or "home" buttons.

2. For the purpose of test generation, we do not need to learn an exact model of the app under test. All we need is an approximate model that can guide the generation

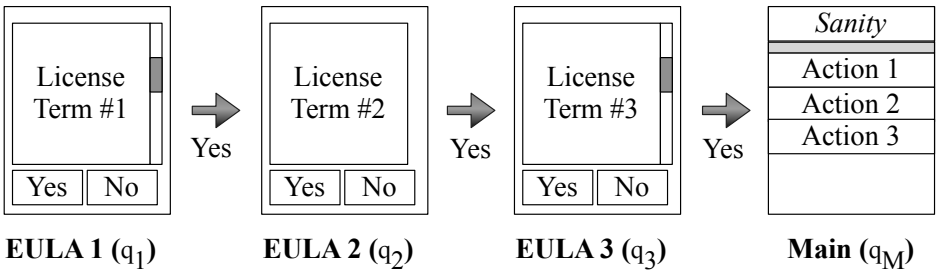


Figure 2.1: The first four screens of the *Sanity* App. The screen names in parentheses are for cross-reference from the models of this app discussed later.

of user inputs while maximizing code coverage. Note that for real-world apps a finite state model of the GUI might not even exist. Some apps will potentially require a push-down model and others a more sophisticated infinite model. For such apps, it is impossible to learn a precise model.

Based on these two observations, we propose a testing algorithm, called SwiftHand, that uses execution traces generated during the testing process to learn an approximate model of the GUI. SwiftHand then uses the learned model to choose user inputs that would take the app to previously unexplored states. As SwiftHand triggers the newly generated user inputs and visits new screens, it expands the learned model, and it also refines the model when it finds discrepancies between the app and the model learned so far. The interplay between on-the-fly learning of the model and generation of user inputs based on the learned model helps SwiftHand to quickly visit unexplored states of the app. A key feature of SwiftHand is that, unlike other learning-based testing algorithms [12, 98], it minimizes the number of restarts by searching for ways to reach unexplored states using only user inputs.

In the rest of the chapter, we will first provide an overview of the SwiftHand algorithm using a running example, and then explain the formal details of the algorithm.

2.1 Overview

In this section we introduce a motivating example, which we will use first to describe several existing techniques for automated user interface testing. We then describe SwiftHand at a high level in the context of the same example. The formal details of the SwiftHand algorithm are in Section 2.2.

We use part of *Sanity*, an Android app in our benchmark-suite, as our running example. Figure 2.1 shows the first four screens of the app. The app starts with three consecutive end-user license agreement (EULA) screens. To test the main app, an automated testing technique must pass all three EULA screens.

The first and the third EULA screens have four input choices: a) **Yes** button to accept the license terms, b) **No** button to decline the license terms, c) **ScrollUp** the license terms, and d) **ScrollDown** the license terms. Pressing **No** at any point terminates program. **ScrollDown** and **ScrollUp** do not change the non-visual state of the program. The second EULA screen doesn't have the scrolling option. Pressing the **Yes** button three times leads the user to the main screen of the app. For convenience, in the remainder of this chapter we are going to use short names q_1, q_2, q_3, and q_M, instead of EULA1, EULA2, EULA2, and Main.

Goals and Assumptions

We want to develop an algorithm that generates sequences of user inputs and feeds them to the app in order to *achieve high code coverage quickly.*

The design of our testing algorithm is guided by the following practical assumptions.

Testing Interface. We assume that it is possible to dynamically inspect the state of the running app to determine the set of user inputs enabled on a given app screen. We also assume that our algorithm can restart the app under test, can send a user input to the app, and can wait for the app to become stable after receiving a user input.

Cost Model. We assume that restarting the app takes significantly more time than sending a user input and waiting for the app to stabilize. Note that a few complicated tasks are performed when an app is restarted: initializing a virtual machine, removing the app's private data, clearing the persistent storage, and executing the app's own initialization code. Testing tools have to wait until the initialization is properly done. Our experiments show that the restart operation takes 30 seconds. Sending a user input is itself very fast, but at the moment our implementation waits for up to 5 seconds for the app to stabilize. We expect that with a more complex implementation, or with some assistance from the mobile platform, we can detect when the handlers have actually finished running. In this case we expect that the ratio of restart cost to the cost of sending a user input will be even higher.

User-Interface Model. We assume that an abstract model of the graphical user interface of the app under test is not available a priori to our testing algorithm. This is a reasonable assumption if we want to test arbitrary real-world Android apps.

When learning a user interface model we have to compare a user interface state with states that are already part of the learned model. For this purpose, we define a notion of state abstraction which enables us to approximately determine if we are visiting the same abstract state. Our abstraction uses a set of enabled user actions inferred from a GUI component tree (e.g., clicking a `CheckBox` type component) and augments each action with some details captured from the GUI component tree (e.g., the checkbox is the third child of the root component, and the box is checked). This means that we do not care about the full

details of GUI components such as colors, coordinates, or text alignment[1]. This abstraction is similar to the one proposed by MacHiry et al. [69]. The details of the abstraction mechanism are available in Chapter 4.

Test Coverage Criteria. We assume that the app under test is predominantly concerned with the user interface, and a significant part of the app state is reflected in the state of the user interface. Thus we assume that a testing approach that achieves good coverage of the user interface states also achieves good code coverage. This assumption is not always entirely accurate, e.g., for apps that have significant internal application state that is not exposed in the user interface.

Existing Approaches

Random Testing. Random testing [69] tests an app by randomly selecting a user input from the set of enabled inputs at each state and by executing the selected input. Random testing also restarts the app at each state with some probability. In the case of *Sanity*, after a restart, random testing has a low probability ($\frac{1}{2} * \frac{1}{2} * \frac{1}{2} = 0.125$) of reaching the main app screen. User inputs that do not change the non-visual state, for example **ScrollUp** and **ScrollDown**, or clicking outside the buttons, do not affect this probability. The expected number of user inputs and restarts required to reach the main app screen are 24 and 7, respectively.[2] This will take about 330 seconds according to our cost model. In summary, random testing has a difficult time on achieving good coverage if an interesting screen is reachable only after executing a specific sequence of user inputs. This is true for our example and is common in real-world apps.

Note that this analysis, and our experiments, use a random testing technique that is aware of the set of enabled user inputs at each state and can make a decision based on this set. A naïve random testing technique, such as the widely-used *Monkey* tester, which

[1] In the original SwiftHand paper [23], an enabled user input is considered according to its type and the bounding box of screen coordinates where it is enabled. We found this abstraction is sensitive to small changes on screen. The new abstraction mechanism, on the contrary, does not depend on the actual coordinate of GUI components and is more tolerant to such small changes.

[2] Let R be the expected number of restarts. At the first EULA screen, if **No** is chosen, the app will terminate and testing should be restarted. This case happens with probability $\frac{1}{4}$. The expected number of restarts is $R_1 = \frac{1}{4}(1 + R)$. If either **ScrollUp** or **ScrollDown** is picked, the app is still in the first EULA screen. Therefore, the expected number of restarts in this case is $R_2 = \frac{1}{2}R$. After considering two more cases (**Yes,No** and **Yes,Yes,Scroll*,No**), we can construct an equation for R.

$$\begin{aligned} R &= R_1 + R_2 + R_3 + R_4 \\ &= \tfrac{1}{4}(1 + R) + \tfrac{1}{2}R + \tfrac{1}{8}(1 + R) + \tfrac{1}{16}(1 + R) \\ &= \tfrac{7}{16} + \tfrac{15}{16}R \end{aligned}$$

Solving the equation, we have $R = 7$. We can perform a similar analysis to get the expected number of all actions including restarts, which is 31. The number of actions except restarts is 24.

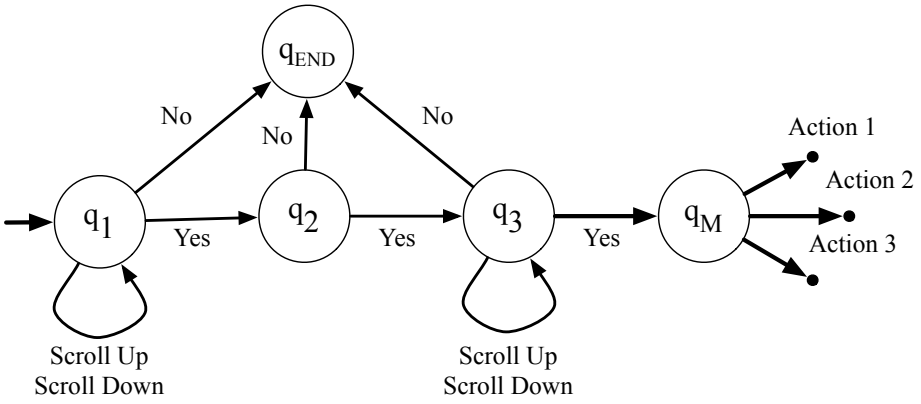


Figure 2.2: A partial model of Android app *Sanity*.

touches random coordinates on the screen, will do nothing meaningful because most of the screen coordinates have no associated event handler.

Model-based Testing. Model-based testing [10,17,78,96,103,113] is a popular alternative to automatically test GUI programs and other event-driven programs. Model-based testing assumes that a finite state model of the GUI is provided by the user. The idea behind model-based testing is to create an efficient set of user input sequences from the model of the target program. The generated test cases could either try to maximize coverage of states or try to maximize coverage of short sequences of user inputs.

Figure 2.2 is a partial model for the *Sanity* app. The model describes a finite state machines. A finite state machine abstracts the infinite state space of the app into a finite number of interesting states and describes equivalence classes of user input sequences leading to those states. A key advantage of using a finite state machine (FSM) model is that an optimal set of test cases can be generated based on a given coverage criterion.

For our running example, if we want to avoid a restart, a model-based testing algorithm could generate the sequence **ScrollDown**, **ScrollUp**, **Yes**, **Yes**, **ScrollDown**, **ScrollUp**, **Yes** to obtain full coverage of non-terminating user inputs and to lead the test execution to the main screen.

Model-based testing can generate optimal test cases for GUI apps, if the model is finite and accurate. Unfortunately, it is a non-trivial task to manually come up with a precise model of the GUI app under test. For several real-world apps, a finite model may not even exist. Some apps could require a push-down automaton or a Turing machine as a model. Moreover, manually generated models may miss transitions that could be introduced by a programmer by mistake.

Testing with Active Learning. Testing with model-learning [41, 88, 91] tries to address the limitations of model-based testing by learning a model of the app as testing is performed.

An active learning algorithm is used in conjunction with a testing engine to learn a model of the GUI app and to guide the generation of user input sequences based on the model.

A testing engine is used as a teacher in active learning. The testing engine executes the app under test to answer two kinds of queries: 1) membership queries—whether a sequence of user inputs is *valid* from the initial state, i.e. if the user inputs in the sequence can be triggered in order, and 2) equivalence queries—whether a learned model abstracts the behavior of the app under test. The testing engine resolves equivalence queries by executing untried scenarios until a counter-example is found. An active learning algorithm repeatedly asks the teacher membership and equivalence queries to infer the model of the app.

However, existing active learning techniques are not suitable to test GUI apps because they perform restart without considering the cost. For example, Angluin's $\mathcal{L}^*$ [12] requires at least $O(n^2)$ restarts, where n is the number of states in the model of the user interface. Rivest and Schapire's algorithm [98] reduces the number of restarts to $O(n)$, which is still high, by computing homing sequences, and it also increases the runtime by a factor of n, which is again not acceptable when we want to achieve code coverage quickly.

A case study: $\mathcal{L}^*$. Angluin's $\mathcal{L}^*$ [12] is the most widely used active learning algorithm for learning finite state machine. The algorithm has been successfully applied to various problem domains from network protocol inference [106] to functional confirmation testing of circuits [66].

We applied $\mathcal{L}^*$ to the running example. We observed that $\mathcal{L}^*$ restarts frequently. Moreover, $\mathcal{L}^*$ made a large number of membership queries to learn a precise and minimal model. Specifically, testing with $\mathcal{L}^*$ required 29 input sequences (i.e. 29 restarts) consisting of 64 user inputs to fully learn the partial model in Figure 2.2. This translates to spending around 870 seconds to restart the app under test (AUT) and 320 seconds for executing the user inputs. 73% of running time is spent on restarting the app. It is important to note that $\mathcal{L}^*$ has to learn the partial model completely and precisely before it can explore the screens beyond the main screen. We show in the experimental evaluation section that $\mathcal{L}^*$ has similar difficulties in actual benchmarks.

Our Testing Algorithm: SwiftHand

SwiftHand combines active learning with testing. However, unlike standard learning algorithms such as $\mathcal{L}^*$, SwiftHand restarts the app under test sparingly. At each state, instead of restarting, SwiftHand tries to extend the current execution path by selecting a user input enabled at the state. SwiftHand uses the model learned so far to select the next user input to be executed.

Informally, SwiftHand works as follows. SwiftHand installs and launches the app under test and waits for the app to reach a stable state. This is the initial *app-state*. For each app-state, we compute a *model-state* based on the set of enabled user inputs in the app-state. Initially the model contains only one state, the model-state corresponding to the initial app-state.

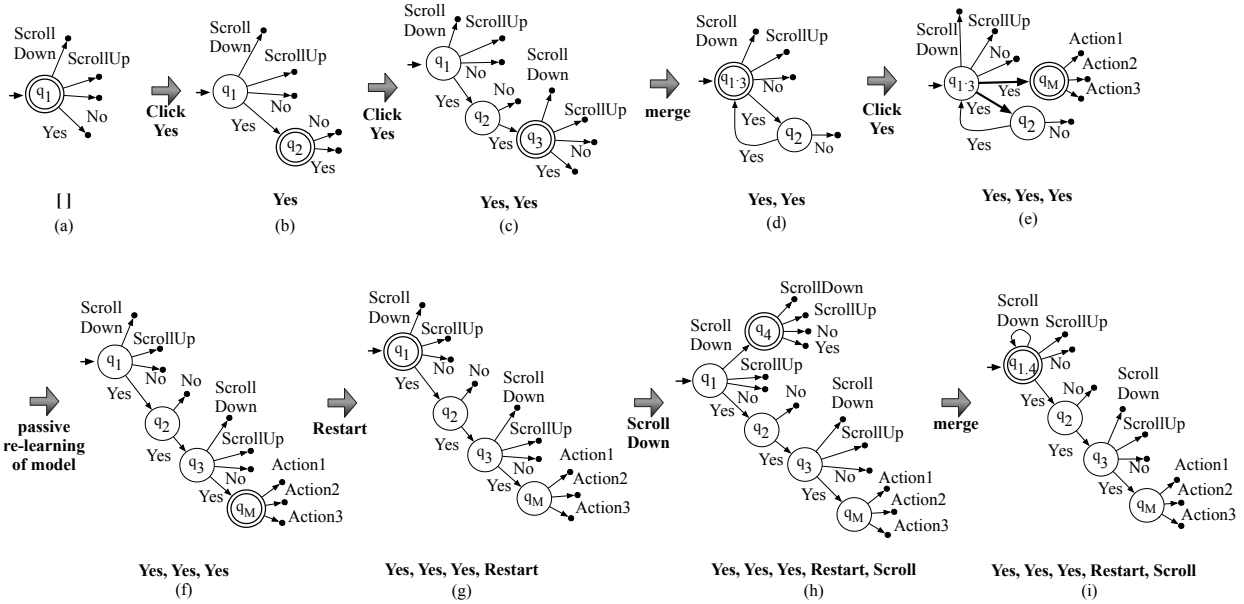


Figure 2.3: Progress of learning-guided testing on the *Sanity* example. The initial state is marked with a short incoming arrow. A solid line with arrow denotes a transition in the model. A circle with a solid line is used to denote a state in the model. A circle with a double line denotes the current model-state.

If a model-state has at least one unexplored outgoing transition, we call it a *frontier model-state*. At each app-state s, SwiftHand *heuristically picks* a frontier model-state q that can be reached from the current model-state without a restart.

Case 0. If such a state is not found, SwiftHand restarts the app. This covers both the case when a frontier-state exists but is not reachable from the current state with user inputs alone, and also the case when there are no frontier-states, in which case our algorithm restarts the app to try to find inconsistencies in the learned model by exploring new paths.

Otherwise, SwiftHand avoids restart by *heuristically finding a path* in the model from the current model-state to q and executes the app along the path. SwiftHand then executes the unexplored enabled input of state q. Three scenarios can arise during this execution.

Case 1. The app-state reached by SwiftHand has a corresponding model-state that has not been encountered before. SwiftHand adds a fresh model-state to the model corresponding to the newly visited app-state and adds a transition to the model.

Case 2. If the reached app-state is equivalent based on enabled user inputs to the screen of a previously encountered app-state, say s', then SwiftHand adds a transition from

q to the model-state corresponding to s'. This is called state *merging*. If there are multiple model-states whose corresponding app-states have equivalent screens, then SwiftHand *picks one of them heuristically for merging*.

Case 3. During the execution of the user inputs to reached the frontier state, SwiftHand discovers that an app-state visited during the execution of the path does not match the corresponding model-state along the same path in the model. This is likely due to an earlier merging operation that was too aggressive. At this point, SwiftHand runs a passive learning algorithm using the execution traces observed so far, i.e. SwiftHand finds the smallest model that can explain the app-states and transitions observed so far.

Note that SwiftHand applies heuristics at three steps in the above algorithm. We discuss these heuristics in Section 2.2. If the target model is finite, SwiftHand will be able to learn the target model irrespective of what heuristics we use at the above three decision points. However, our goal is not necessarily to learn the exact model, but to achieve coverage quickly. Therefore, we apply heuristics that enable SwiftHand to explore previously unexplored states quickly. In order to avoid SwiftHand from getting stuck in some remote part of the app, we allow SwiftHand to restart when it has executed a predefined number of user inputs from the initial state.

Figure 2.3 illustrates how SwiftHand works on the running example. For this example, we pick user inputs carefully in order to keep our illustration short, yet comprehensive. For this reason we do not pick the **No** user input to avoid restart. In actual implementation, we use various heuristics to make such decisions. A model-state with at least one unexplored outgoing transition is called a *frontier* state. A solid line with arrow denotes a transition in the model. The input sequence shown below the diagram of a model denotes an input sequence of the current execution from the initial state.

- *Initialization*: After launching the app, we reach the initial app-state, where the enabled inputs on the state are {**Yes**, **No**, **ScrollUp**, **ScrollDown**}. We abstract this app-state as the model-state q_1 (using the same terminology as in Figure 2.1 and Figure 2.2). This initial state of the model is shown in Figure 2.3(a).

- *1st Iteration*: Starting from the initial state of the model, SwiftHand finds that the state q_1 is a frontier state and chooses to execute a transition for the input **Yes**. The resulting state has a different set of enabled inputs from the initial state. Therefore, according to Case 1 of the algorithm, SwiftHand adds a new model-state q_2 to the model. The modified model is shown in Figure 2.3(b).

- *2nd Iteration*: The app is now at an app-state whose corresponding model-state is q_2, as shown in Figure 2.3(b). Both q_1 and q_2 have unexplored outgoing transitions. However, if we want to avoid restart, we can only pick a transition from q_2 because according to the current model there is no sequence of user inputs to get to q_1 from the

current model-state. SwiftHand chooses to execute a transition on **Yes**, and obtains a new app-state for which it creates a new model-state q_3, as shown in Figure 2.3(c). However, the new app-state has the same set of enabled inputs as the initial app-state q_1. Therefore, SwiftHand merges q_3 with q_1 according to Case 2. This results in the model shown in Figure 2.3(d). If you compare the partial model learned so far with the actual underlying model shown in Figure 2.2, you will notice that the merging operation is too aggressive. In the actual underlying model the state reached after a sequence of two **Yes** clicks is different than the initial state. This will become apparent to SwiftHand once it explores the app further.

- *3rd Iteration*: The app is now at an app-state whose corresponding model-state is q_1, as shown in Figure 2.3(d). SwiftHand can now pick either q_1 or q_2 as the next frontier state to explore. Assume that SwiftHand picks q_2 as the frontier state to explore. A path from the current model-state q_1 to q_2 consists of a single transition **Yes**. After executing this input, however, SwiftHand encounters an inconsistency—the app has reached the main screen after executing the input sequence **Yes**,**Yes**,**Yes** from the initialization (see Figure 2.2). In the current model learned so far (Figure 2.3(d)), the abstract state after the same sequence of actions ought to be q_2. Yet the set of enabled inputs associated with this screen (**Action1**, **Action2**, and **Action3**) is different from the set of enabled inputs associated with q_2.

  We say that SwiftHand has discovered that the model learned so far is inconsistent with the app, and the input sequence **Yes**,**Yes**,**Yes** is a counter-example showing this inconsistency. Figure 2.3(e) illustrates this situation; notice that there are two outgoing transitions labeled **Yes** from state q_2. This is Case 3 of the algorithm. The inconsistency happened because of the merging decision made at the second iteration. To resolve the inconsistency, SwiftHand abandons the model learned so far and runs an off-the-shelf passive learning algorithm [63] to rebuild the model from scratch using all execution traces observed so far. Figure 2.3(f) shows the result of passive learning. Note that this learning is done using the transitions that we have recorded so far, without executing any additional transitions in the actual app.

- *4th Iteration*: SwiftHand is forced to restart the app when it has executed a predefined number of user inputs from the initial state. Assume that SwiftHand restarts the app in the 4th iteration so that we can illustrate another scenario of SwiftHand. After restart, q_1 becomes the current state (see Figure 2.3(g)). SwiftHand now has several options to execute an unexplored transition. Assume that SwiftHand picks **ScrollDown** transition out of the q_1 state for execution. Since scrolling does nothing to the screen state and the set of enabled inputs, we reach an app-state that has the same screen and enabled inputs as q_1 and q_3. SwiftHand can now merge the new model-state with either q_1 or q_3 (see Figure 2.3(h)). In practice, we found that the nearest ancestor or the nearest state with the same set of enabled inputs works better

than other model-states. We use this heuristics to pick q_1. The resulting model at this point is shown in Figure 2.3(i).

After the first four iterations, SwiftHand will execute 4 restarts and 11 more user inputs, in the worst case, to learn the partial model in Figure 2.2. The restarts are necessary to learn the transitions to the terminal state, `End`. If SwiftHand wants to explore states beyond the main screen after a restart, it can consult the model and execute the input sequence **Yes**, **Yes**, and **Yes** to reach the main screen. Random testing will have a hard time reaching the main screen through random inputs. In terms of our cost model, SwiftHand will spend 190 seconds or 60% of the execution time in restarting the app. The percentage of time spent in restarting drops if the search space becomes larger. In our actual benchmarks, we observed that SwiftHand spent about 10% of the total time in restarting.

2.2 Learning-guided Testing Algorithm

In this sections, we formally describe the SwiftHand algorithm. We first introduce a few definitions that we use in our algorithm. We then briefly describe the algorithm. SwiftHand uses a variant of an existing passive learning algorithm to refine a model whenever it observes any inconsistency between the learned model and the app. We describe this passive learning algorithm to keep the section self-contained.

Models as ELTS. We use *extended deterministic labeled transition systems* (ELTS) as models for GUI apps. An ELTS is a deterministic labeled transition system whose states are labeled with a set of enabled transitions (or user inputs). Formally, an ELTS M is a tuple

$$M = (Q, q_0, \Sigma, \delta, \lambda)$$

where

- Q is a set of states,
- $q_0 \in Q$ is the initial state,
- Σ is an input alphabet,
- $\delta : Q \times \Sigma \to Q$ is a partial state transition function,
- $\lambda : Q \to \wp(\Sigma)$ is a state labeling function. $\lambda(q)$ denotes the set of inputs enabled at state q, and
- for any $q \in Q$ and $a \in \Sigma$, if there exists a $p \in Q$ such that $\delta(q, a) = p$, then $a \in \lambda(q)$.

The last condition implies that if there is a transition from q to some p on input a, then a is an enabled input at state q.

Arrow. We use $q \xrightarrow{a} p$ to denote $\delta(q, a) = p$.

Arrow*. We say $q \xrightarrow{l} p$ where $l = a_1, \ldots, a_n \in \Sigma^*$ if there exists $q_1, \ldots, q_{n-1}$ such that $q \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \ldots q_{n-1} \xrightarrow{a_n} p$.

Trace. An execution trace, or simply a trace, is a sequence of pairs of inputs and sets of enabled inputs. Formally, a trace t is an element of $(\Sigma \times \wp(\Sigma))^*$. In this section, we use this simplified trace definition for the clarity of the presentation.

Trace Projection. We use $\pi(t)$ to denote the sequence of inputs in the trace t. Formally, if $t = (a_1, \Sigma_1), \ldots, (a_n, \Sigma_n)$, then $\pi(t) = a_1, \ldots, a_n$.

Arrow Trace. We say $q \xrightarrow{t} p$ if $q \xrightarrow{\pi(t)} p$.

Consistency. A trace $t = (a_1, \Sigma_1), \ldots, (a_n, \Sigma_n)$ is consistent with a model $M = (Q, q_0, \Sigma, \delta, \lambda)$ if and only if

$$\exists q_1, \ldots q_n \in Q. \bigwedge_{i \in [1,n]} q_{i-1} \xrightarrow{a_i} q_i \wedge \lambda(q_i) = \Sigma_i.$$

Frontier State. A state in an ELTS is a *frontier-state* if there is no transition from the state on some input that is enabled on the state. Formally, a state q is a frontier-state if there exists a $a \in \lambda(q)$ such that $q \xrightarrow{a} p$ is not true for any $p \in Q$.

Terminal State. When the execution of an app terminates, we assume that it reaches a state that has no enabled inputs.

Learning-guided Testing Algorithm

Interface with the App under Test. SwiftHand treats the app state as a black box. However, it can query the set of enabled inputs on an app state. We assume that $\lambda(s)$ returns the set of enabled inputs on an app state s. SwiftHand can also ask the app to return a new state and trace after executing a sequence of user inputs from a given app state. Let s be an app state, t be a trace of executing the app from the initial state s_0 to s, and l be a sequence of user inputs. Then Execute(s, t, l) returns a pair containing the app state after executing the app from state s on the input sequence l and the trace of the entire execution from the initial state s_0 to the new state.

Algorithm 1 SwiftHand: Learning-guided testing algorithm

```
1: procedure Testing(s_0)                                   ▷ s_0 is the initial state of the app
2:     M ← ({q_0}, q_0, Σ, ∅, {q_0 ↦ λ(s_0)}) for some fresh state q_0   ▷ M is the current model
3:     T ← ∅                                                ▷ T accumulates the set of traces executed so far
4:     p ← q_0 and s ← s_0 and t ← ϵ
5:                                  ▷ p, s, and t are the current model-state, app-state, and trace, respectively
6:     while ¬timeout() do                                  ▷ While time budget for testing has not expired
7:         if |t| > MAX_LENGTH then                         ▷ Current trace is longer than a maximum limit
8:             p ← q_0 and s ← s_0 and T ← T ∪ {t} and t ← ϵ          ▷ Restart the app
9:         else if there exists a frontier-state q in M then          ▷ Model is not complete yet
10:            if there exists l ∈ Σ* and a ∈ Σ such that p —l→ q and a ∈ λ(q) then
11:                (s, t) ← Execute(s, t, l)
12:                if t is consistent with M then
13:                    (s, t) ← Execute(s, t, a)
14:                    if there exists a state r in M such that λ(r) = λ(s) then   ▷ Merge with an existing state
15:                        add q —a→ r to M
16:                        p ← r                            ▷ Update current model-state
17:                    else                                 ▷ Add a new model-state
18:                        add a fresh state r to M such that q —a→ r and λ(r) = λ(s)
19:                        p ← r                            ▷ Update current model-state
20:                    end if
21:                else                                     ▷ Inconsistent model. Need to re-learn the model.
22:                    T ← T ∪ {t} and M ← PassiveLearn(T)
23:                    p ← r where q_0 —t→ r                ▷ Update current model-state
24:                end if
25:            else                                         ▷ A frontier-state cannot reached from the current state
26:                p ← q_0 and s ← s_0 and T ← T ∪ {t} and t ← ϵ          ▷ Restart the app
27:            end if
28:        else if there exists state q in M and l ∈ Σ* such that p —l→ q and l is not a subsequence of π(t) for any t ∈ T then
29:                                                          ▷ Model is complete, but may not be equivalent to target model
30:            (s, t) ← Execute(s, t, l)
31:            if t is not consistent with M then           ▷ Inconsistent model. Need to re-learn the model.
32:                T ← T ∪ {t} and M ← PassiveLearn(T)
33:                p ← r where q_0 —t→ r                    ▷ Update current model-state
34:            end if
35:        else                                             ▷ Model is complete and equivalent to target model
36:            return T                                     ▷ Done with learning
37:        end if
38:    end while
39:    return T
40: end procedure
```

Description of the Actual Algorithm. The pseudo-code of the algorithm is shown in Algorithm 1. The algorithm maintains five local variables: 1) s denotes the current app-state and is initialized to s_0, the initial state of the app, 2) p denotes the current model-state, 3) t denotes the current trace that is being executed, 4) T denotes the set of traces tested so far, and 5) M denotes the ELTS model learned so far.

At each iteration the algorithm tries to explore a new app state. To do so, it finds a frontier-state q in the model, then finds a sequence of transitions l that could lead to the frontier-state from the current model-state, and a transition a enabled at the frontier-state (lines 9–10). It then executes the app on the input sequence l from the current app state s and obtains a trace of the execution (line 11). If the trace is not consistent with the model, then we know that some previous merging operation was incorrect and we re-learn the model

using a passive learning algorithm from the set of traces observed so far (lines 21–24). On the other hand, if the trace is consistent with the model learned so far, the algorithm executes the app on the input a from the latest app state (lines 12–13). If the set of enabled inputs on the new app state matches with the set of enabled inputs on an existing model-state (line 14), then it is possible that we are revisiting an existing model-state from the frontier state q on input a. The algorithm, therefore, merges the new model-state with the existing model-state (line 15). Note that this is an approximate check of equivalence between two model-states, but it helps to prune the search space. If the algorithm later discovers that the two states are not equivalent, it will perform a passive learning to learn a new model, effectively undoing the merging operation. Nevertheless, this aggressive merging strategy is key to prune similar states and guide the app into previously unexplored state space. On the other hand, if the algorithm finds that the set of enabled inputs on the new app-state is not the same as the set of enabled inputs of any existing model-state, then we have visited a new app-state. The algorithm then adds a new model-state corresponding to the app-state to the model (line 18). In either case, that is whether we merge or we add a new model-state, we update our current model-state to the model-state corresponding to the new app state and repeat the iteration (lines 16 and 19).

During the iteration, if we fail to find a frontier state, we know that our model is *complete*, i.e. every transition from every model-state has been explored. However, there is a possibility that some incorrect merging might have happened in the process. We, therefore, need to now confirm that the model is equivalent to the target model of the app. This is similar to the equivalence check in the $\mathcal{L}^*$ algorithm. The algorithm picks a sequence of transitions l such that l is not a subsequence of any trace that has been explored so far (lines 28–30). Moreover, l should lead to a state q from the current state in the model. If such an l is not found, we know that our model is equivalent to the target model (lines 35–36). On the other hand, if such an l exists, the algorithm executes the app on l and checks if the resulting trace is consistent with the model (lines 30–31). If an inconsistency is found, the model is re-learned from the set of traces executed so far (lines 32–33). Otherwise, we continue the refinement process over any other existing l.

During the iteration, if the algorithm finds a frontier state, but fails to find a path from the current state to the frontier state in the model, it restarts the app (lines 25–26). We observed that in most GUI apps, it is often possible to reach most screen states from another screen state after a series of user inputs while avoiding a restart. As such, this kind of restart is rare in SwiftHand. In order to make sure that our testing does not get stuck in some sub-component of an app with a huge number of model-states, we do restart the app if the length of a trace exceeds some user defined limit (i.e. MAX_LENGTH) (lines 7–8).

Heuristics and Decision Points

The described algorithm has five decision points and we use the following heuristics to resolve them. We came up with these heuristics after extensive experimentation.

- If there are multiple frontier-states and if there are multiple enabled inputs on those frontier-states (lines 9–10), then which frontier-state and enabled transition should we pick? SwiftHand picks a random state from the set of frontier-states and picks a random transition from the frontier state. We found through experiments that the effectiveness of SwiftHand does not depend on what heuristics we use for this case.

- If there are multiple transition sequences l from the current model-state to a frontier state (line 10), which one should we pick? We found that picking a short sequence is not necessarily the best strategy. Instead, SwiftHand selects a sequence of transitions from the current model-state to the frontier state so that the sequence contains a previously unexplored sequence of inputs. This helps SwiftHand to learn a more accurate model early in the testing process.

- If multiple states are available for merging (at line 14), then which one should SwiftHand pick? If we pick the correct model-state, we can avoid re-learning in future. We experimented with a random selection strategy and with a strategy that selects a nearby state. However, we discovered after some experimentation that if we prefer the nearest ancestor to other states, our merge operations are often correct. Therefore, SwiftHand uses a heuristics that first tries to merge with an ancestor. If an ancestor is not available for merging, SwiftHand picks a random state from the set of candidate states.

- If there are multiple transition sequences l available for checking equivalence (line 28), which one should we pick? In this case, we again found that none of the strategies we tried make a difference. We therefore use random walk to select such an l.

- We set the maximum length of a trace (i.e. MAX_LENGTH) to 50. We again found this number through trial-and-error.

Rebuilding a Model using Passive Learning.

We describe the passive learning algorithm that we use for re-learning a model from a set of traces. The algorithm is a variant of Lambeau et al.'s [63] state-merging algorithm. We have modified the algorithm to learn ELTS. We describe this algorithm to keep the section self-contained.

Prefix Tree Acceptor. A prefix tree acceptor [11] (or a PTA) is an ELTS whose state transition diagram is a tree with the initial state of the ELTS being the root of the tree. Given a set of traces T, we build a prefix tree acceptor PTA_T whose states are the set of all prefixes of the traces in T. There is a transition with label a from t to t' if t can be extended to t' using the transition a. The $\lambda(t)$ is Σ' if the last element of t has Σ' as the second component.

Partitioning and Quotient Model. $\Pi \subseteq \wp(Q)$ is a partition of the state space Q if all elements of Π are disjoint, all elements of Q are a member of some element of Π, and λ of all elements of a given element of Π are the same. An element of Π is called an equivalence class and is denoted by π. $\texttt{element}(\pi)$ denotes a random single element of π. M/Π is a quotient model of M obtained by merging equivalent states with respect to Π:

$$\frac{\begin{array}{c}\pi_0 \text{ is the partition containing } q_0 \\ \delta' \stackrel{\text{def}}{=} \{(\pi, a) \mapsto \pi' \mid \exists q \in \pi \text{ and } \exists q' \in \pi' \,.(q, a) \mapsto q' \in \delta\} \\ \forall \pi \in \Pi.\lambda'(\pi) \stackrel{\text{def}}{=} \lambda(\texttt{element}(\pi))\end{array}}{M/\Pi = (\Pi, \pi_0, \Sigma, \delta', \lambda')}$$

Note that a quotient model can be non-deterministic even though the original model is deterministic.

The Algorithm. Algorithm 2 describes the state-merging based learning algorithm. Conceptually, the algorithm starts from a partial tree acceptor PTA_T and repeatedly generalizes the model by merging states. The merging procedure first checks whether any two states agree on the λ function, and then tries to merge them. If merging results in a non-deterministic model, the algorithm tries to eliminate non-determinism by merging target states of non-deterministic transitions provided that the merged states have the same λ. This process is applied recursively until the model is deterministic. If at some point of the procedure, merging of two states fails because λs of the states are different, the algorithm unrolls the entire merging process for the original pair of states.

The ChoosePair procedure decides the order of state merging. The quality of the learned model solely depends on the implementation of this function. We will explain the procedure in short.

Our algorithm differs from the original algorithm on two fronts. First, the original algorithm [63] aims to learn DFA with mandatory merging constraints and blocking constraints. Our algorithm learns an ELTS and only uses the idea of blocking constraints. We use the λ function or the set of enabled transitions at any state to avoid illegal merging. Second, DFA learning requires both positive and negative examples. ELTS has no notion of negative examples.

Selecting a pair of equivalence classes. Algorithm 3 describes our algorithm for selecting a pair of model-states to merge. The ChoosePair procedure selects a pair of model-states such that merging the pair would minimize the number of states in the model after the merge. This strategy is known as evidence-driven state-merging [28, 64].

The procedure works by constructing a set of candidate pairs (line 2-3) and evaluating each pair by simulating the merge (lines 3-10). The procedure stores the evaluation result of each pair in table f, which maps a pair of states to the size of the model after merging the pair. If a pair can be merged successfully, the table will update (line 8). After constructing

Algorithm 2 Passive learning algorithm

```
1: procedure REBUILD(T)
2:     Π ← {{q} | q ∈ Q_{PTA_T}}
3:     while (π_i, π_j) ← CHOOSEPAIR(Π) do
4:         Try
5:             Π ← MERGE(Π, π_i, π_j)
6:         CatchAndIngnore
7:     end while
8:     return PTA_T/Π
9: end procedure

10: procedure MERGE(Π, π_i, π_j)
11:     M ← PTA_T/Π
12:     if λ_M(π_i) ≠ λ_M(π_j) then
13:         throw exception
14:     end if
15:     π_pivot ← π_i ∪ π_j
16:     Π ← (Π \ {π_i, π_j}) + π_pivot
17:     while (π_k, π_l) ← FINDNONDETER(Π, π_pivot) do
18:         Π ← MERGE(Π, π_k, π_l)
19:     end while
20:     return Π
21: end procedure

22: procedure FINDNONDETER(Π, π)
23:     M ← PTA_T/Π
24:     S ← {(π_i, π_j) | ∃a ∈ λ_M(π).π -a->_M π_i ∧ π -a->_M π_j ∧ π_i ≠ π_j}
25:     return pick(S)
26: end procedure
```

the evaluation table, the procedure returns a candidate pair that minimizes the number of states if the table is not empty (line 14). However, if the table is empty the procedure returns null because this indicates that no two states can be merged (line 12).

It would be ideal to consider all possible pairs of model-states, as this provides an optimal solution. However, only a small portion of state pairs is worth considering [64], so considering all possible pairs is inefficient. As such, the procedure applies the `ChooseCandidates` function to select a subset of candidate pairs using the BlueFringe [64] method, generally considered as the best-known heuristic.

Optimizations. In practice, a straightforward implementation of the passive learning algorithm (Algorithm 2 and Algorithm 3) can be a bottleneck if the SWIFTHAND algorithm is executed for several hours. Note the worst case time complexity of the passive learning

Algorithm 3 Selecting a pair of equivalence classes to merge

```
1: procedure ChoosePair(Π)
2:     P ← {(π_i, π_j) | π_i, π_j ∈ Π ∧ π_j ≠ π_j ∧ λ(element(π_i)) = λ(element(π_j))}
3:     C ← ChooseCandidates(P)
4:     f ← [ ]
5:     for (π_i, π_j) ∈ C do
6:         Try
7:             Π' ← Merge(Π, π_i, π_j)
8:             f ← f + [(π_i, π_j) ↦ |Π'|]
9:         CatchAndIngnore
10:    end for
11:    if dom(f) = ∅ then
12:        return null
13:    else
14:        return argmin_{(π_i,π_j) ∈ dom(f)} f(π_i, π_j)
15:    end if
16: end procedure
```

algorithm is $O(n^3)$ where n the size of the initial partial tree acceptor PTA_T. The passive learning algorithm calls the Merge procedure $O(n^2)$ times and the execution time of each Merge call is linear in n. When SwiftHand runs for several hours, PTA_T could incorporate tens of thousands of states. The cubic time complexity is unacceptable in such a case.

The time complexity of the algorithm could be decreased using dynamic programming:

- Within a single Rebuild procedure call, we can maintain a set of candidate pairs that cannot be merged. Note that if two model-states π_i and π_j are identified as non-mergeable, then they will remain non-mergeable throughout the rest of the execution of the same Rebuild call. Therefore, if a pair of model-states is identified as non-mergeable, we can bookkeep the unavailability and never check that particular pair again in the same Rebuild call.

- We can also accelerate the Merge procedure by collecting and reusing information across multiple Rebuild procedure calls. Assume that a pair of model-states cannot be merged in one instance of Rebuild call. This means that there is a counter-example (a sequence of actions) revealing a difference between two model-states. The same sequence might prevent other pairs of model-states from being merged in a future Rebuild call. As such, we can maintain a list of counter-examples for merge failures and check this list before calling the Merge procedure. Modern active-learning algorithms [54, 60] have used a discriminator tree, a technique based on the same observation, to reduce the number of membership queries.

Chapter 3

DetReduce, a GUI Test Suite Minimization Algorithm

In the previous chapter we introduced SwiftHand, an automated GUI testing algorithm for Android apps. It works by injecting sequences of automatically generated user actions to an app. Each sequence of actions injected by the algorithm can be seen as a *test case*, and the set of all sequences of actions can be considered as a *test suite*. Although SwiftHand achieves good code and screen coverage (i.e. covering all distinct screens of an app), as we will see in Chapter 5, it must run for several hours to saturate branch coverage and screen coverage. As a result, SwiftHand typically generates test suites containing thousands of test cases—each of which could contain tens to thousands of user actions. Such a large number of test cases cannot be used for regression testing because regression testing would then take the same time as running the actual automated testing tool. Regression tests need to run faster so that they can be used frequently during development.

In this chapter, we address the problem of generating a small regression GUI test suite for an Android app. We assume that we are given a large test suite generated by an existing automated GUI testing tool, such as SwiftHand. We assume that the test suite is replayable in the sense that if we rerun the test suite multiple times we will achieve the same coverage and observe the same sequence of app screens. A test suite directly produced by a test generation tool can contain non-replayable test cases. However, we can find a replayable prefix of the test case by executing the same test case multiple times (ten in our evaluation). By repeating this process for all test cases in a test suite, we obtain a test suite composed of replayable prefixes of the test cases in the original test suite[1]. A replayable test suite obtained in this way, however, can take several hours to run on the app. Our goal is to spend a reasonable amount of time—perhaps one day— to generate a small regression test suite for the app that runs for less than an hour and that achieves similar code and screen coverage similar to that of the original test suite.

[1] Section 5.3 explains how to determine if a test case is replayable and how to obtain a replayable test suite from an automated GUI testing tool.

Several techniques have been proposed to minimize test suites for GUIs. For example, Clapp et al. [25] proposed a delta-debugging [124] based algorithm that can minimize the test suite of a GUI app while tolerating non-determinism. Hammoudi et al. [45] proposed another delta-debugging technique to minimize manually-written test suites for web applications. These existing techniques work well if the size of the input test suite is small, containing fewer than one thousand user inputs. However, they do not scale for large test suites because these techniques depend heavily on the rapid generation and feasibility checking of new test cases. Unfortunately, for most real-world GUI apps it takes several minutes to check the feasibility of a new input sequence. For large test suites containing tens of thousands of user actions, it can therefore take more than a month to effectively minimize a test suite. McMaster and Memon [74] proposed a technique [118] for reducing the number of test cases in a test suite, but this technique does not reduce the size of each test case. In our experimental evaluation, we observed that test cases generated by an automated tool can contain subsequences of redundant user actions, which can be removed to obtain smaller test suites.

We propose a GUI test suite reduction algorithm that can scalably and effectively minimize large test suites. The key insight behind our technique is that if we can identify and remove common forms of redundancies introduced by existing automated GUI testing tools, then we can markedly lower the time required to minimize a GUI test suite. We manually analyzed several sources of redundancies in the test suites generated by the automated GUI testing tool SwiftHand, and identified three kinds of redundancies that are common in these test suites: 1) some test cases can be safely removed from a test suite without decreasing code and screen coverage, 2) within a test case, certain loops can be eliminated without decreasing coverage, and 3) many test cases share common subsequences of actions whose repeated execution can be avoided by combining fragments from different action sequences. Based on these observations, we have developed an algorithm that removes such redundancies one-by-one while ensuring that we do not reduce the overall code and screen coverage of the resulting test suite.

In order to identify redundant loops and common subsequences of test cases, we define a notion of state abstraction that enables us to approximately determine if we are visiting the same abstract state at least twice while executing a test case. If an abstract state is visited twice during the execution, we have identified a loop that can potentially be removed. Similarly, if the execution of two test cases visits a subsequence of the same abstract states, we know that fragments from the two test cases can be combined to obtain a longer trace that avoids executing the common fragment. However, when we create a new test case by removing a loop or combining two fragments, the resulting test case might not traverse the same abstract states as expected. In our algorithm, we check the feasibility of a newly created test case by executing it ten times and determining if the execution visits the same sequence of abstract states every time—we call this replayability. We have observed that if our state abstraction is too coarse-grained, feasibility checks fail often, leading to longer running time of the algorithm. On the other hand, if we use a too fine-grained state abstraction, we fail to identify many redundancies. One contribution of this chapter is to put forward a sufficiently

effective abstraction that works well in practice. Our algorithm spends most of its time checking the feasibility of newly created test cases; if such a check fails for a new test case, we remember the prefix of the test case that failed and use it to avoid checking the feasibility of any future test case with the same prefix. This simple pruning helps us to reduce the number of new test cases to check.

One key advantage of our algorithm over delta-debugging or other black-box oriented algorithms is that we do not blindly generate all possible new test cases that can be generated by dropping some actions. Rather, we apply a suitable state abstraction to drop only redundant loops. Another key advantage is that we create new test cases by combining fragments from input test cases, which enables us to produce new, longer test cases that cannot be generated using delta-debugging. Longer test cases are generally superior to multiple shorter test cases because we need not perform a clean restart of the app. A clean restart of an app is timing-consuming, because it requires killing the app, erasing app data, and erasing SD card contents. A longer test case in place of several shorter test cases avoids several such expensive restarts.

3.1 Overview

In this section, we give a brief overview of our technique using formal notation and a series of examples. The formal details of our technique are in Section 3.2

Definitions and Problem Statement

We introduce a few simple definitions and describe the problem that we are solving.

Trace. The execution of an app on a sequence of user inputs can be denoted by a *trace* of the form

$$s_0 \xrightarrow{a_1,C_1} s_1 \xrightarrow{a_2,C_2} \ldots \xrightarrow{a_n,C_n} s_n$$

where

- each s_i is an abstract state of the program, usually computed by abstracting the screen of the app,
- each $s_{i-1} \xrightarrow{a_i,C_i} s_i$ is a *transition* denoting that the app transitioned from state s_{i-1} to state s_i on user input (or action) a_i and C_i is the set of branches covered during the handling of the action by the app. Here we focus on branch coverage; however, one could use other kinds of coverage for C_i.

Coverage. If $s_{i-1} \xrightarrow{a_i,C_i} s_i$ is a transition, then $C_i \cup \{s_i\}$ is the *coverage of the transition*. In the coverage we include both the set of branches and the abstract states visited by the transition. We can similarly define the coverage of a trace $\tau = s_0 \xrightarrow{a_1,C_1} s_1 \xrightarrow{a_2,C_2} \ldots \xrightarrow{a_n,C_n} s_n$

as the union of the coverage of all the transitions in the trace, i.e. $\cup_{i\in[1,n]}(C_i \cup \{s_i\})$ and denote it by $C(\tau)$.

Replayable traces. In our technique, we are only interested in *replayable* traces. A trace $\tau = s_0 \xrightarrow{a_1,C_1} s_1 \xrightarrow{a_2,C_2} \ldots \xrightarrow{a_n,C_n} s_n$ of an app is *replayable*, if every time the app is given the sequence of user actions $a_1, a_2, \ldots, a_n$ in a state s_0, it generates the exact trace τ.

Test suite: A set of replayable traces. We assume that an automated testing tool for GUI generates *a set T_s of replayable traces* that can be treated as a regression test suite. Our evaluation section describes a methodology to obtain a replayable set of traces from SwiftHand. The *coverage of a set of traces* T, denoted by $C(T)$, is defined as the union of the coverage of the traces contained in the set. The *cost of a set of traces* T is the pair $(\sum_{\tau\in T}|\tau|, |T|)$. The first component of the pair gives the number of transitions present in the traces in T. This number roughly estimates the amount of time necessary to replay the traces in T. Between the replay of two traces, one needs to perform a clean restart of the app by erasing the app data and SD card contents, which has high cost. In order to take that cost into account, we have a second component in the pair denoting the number of clean restarts necessary to replay all the traces in T.

Problem statement. Given a set of replayable traces T_s, the goal of our technique is to find a minimal set of traces T_0 such that T_0 is replayable, T_0 consists of transitions from the traces in T_s, $C(T_s) = C(T_0)$, and the cost of T_0 is minimal.

Unfortunately, finding a minimal T_0 is intractable in practice. First, without the replayability requirement, the problem can be reduced to an instance of the prize-collecting traveling salesman problem (PCTSP), a well-known NP-hard problem [16]. Unfortunately, with the replayability requirement, a solution found by solving the corresponding PCTSP problem could include non-replayable traces. Therefore, we need to solve multiple PCTSP problems until we finally find a replayable solution. This makes the problem intractable in practice. Instead of solving the problem of finding the global minimum, we developed a two-phase heuristic algorithm, which we found to work effectively in practice.

Limitations of Existing Approaches

In any test-suite reduction technique, we need to construct new traces. Although the creation of a trace takes minimal time, we have to ensure that the trace can be replayed. It is impossible to precisely determine if a trace is replayable. In our technique, we check if a trace is replayable by executing it few times (which is ten times in our experimental evaluation). We found experimentally that if a trace is non-replayable, it will fail to replay within ten executions. Faithfully executing a single transition in a trace could take a few seconds because after injecting an input or action, we need to wait until the screen stabilizes. Therefore, executing a trace composed of several transitions could take several minutes.

Moreover, after executing each trace we need to erase app data and SD card contents before a clean restart. This action takes several seconds. Therefore, it is generally time consuming to check if a trace is replayable. This is the key bottleneck faced by a GUI test suite reduction technique.

Existing test minimization techniques, such as delta-debugging [124] and evolutionary-algorithms [70], will create and check the replayability of lots of traces. Therefore, these techniques cannot scale when the initial set of traces is large. Unfortunately, all existing automated techniques for GUI test generation create lots of traces. Clapp et al. [25] recently proposed a delta-debugging based GUI test minimization technique. Their experimental results show that their technique can take a few hours to several tens of hours to handle traces composed of only 500 transitions. In our experiments, we had to handle test suites having 10,000 transitions. If we extrapolate the timings reported by Clapp et al. to 10,000 transitions, delta-debugging will easily take a month. We want a technique that can minimize a test suite in a day or less.

Our Observations

We observed that the set of traces generated by an automated testing tool has lots of redundancies. Our technique for GUI test suite reduction tries to remove these redundancies using a heuristic algorithm. We next describe these redundancies using a series of examples.

Redundant Traces

Among the traces in a test suite, the coverage provided by some traces is a subset of coverage provided by the remaining traces. Such traces can be removed from the test suite without decreasing the cumulative coverage. Our technique greedily finds a large set of such redundant traces and removes them from the test suite.

Redundant Loops

We also observed that there could be redundancies within a trace, for example, if the trace contains a *redundant loop*. A loop in a trace is a sub trace of the trace that begins and ends in the same abstract state. Traces generated by automated testing tools tend to contain many loops, and some such loops do not provide additional coverage over the coverage that can be achieved by the trace without the loop and the remaining traces. Such loops are redundant and can be potentially removed from the trace if the resulting trace can be replayed. We next illustrate such redundant loops using a couple of examples. All examples utilize the file browser app shown in Figure 3.1.

Example 3.1. (A redundant loop)

Assume that the user touches the menu button three times. This input sequence will open the pop-up menu, close it, and then open it again. The user then selects the first item (i.e. the Option button) of the menu. This action will lead the app to the configuration

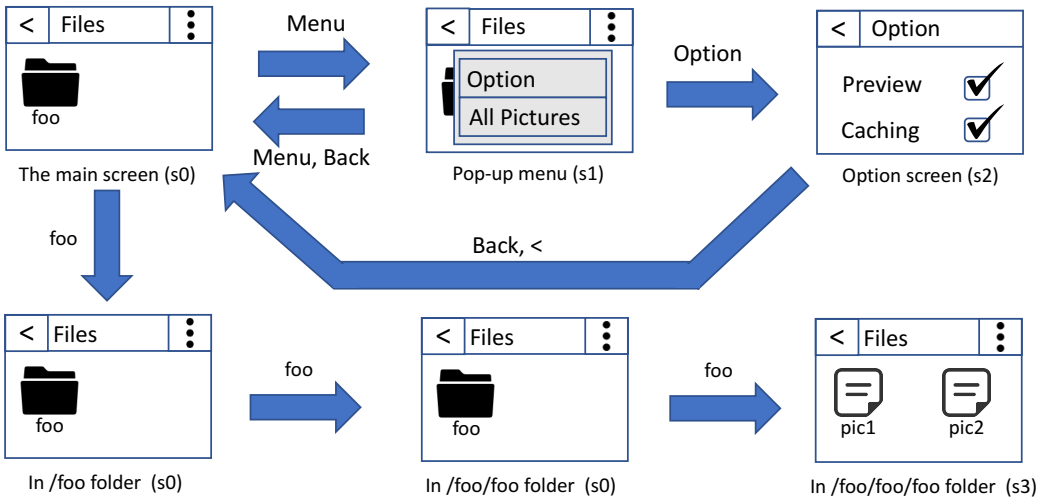


Figure 3.1: A partial model of a file browser app. The main screen of the app is a file system browser. When the app starts, it shows the root directory (abstract state/screen s_0). In this initial state/screen, a user can invoke a pop-up menu (abstract state/screen s_1) by touching a menu button on the screen (the button at the top-right corner of the screen with three dot characters). Once the pop-up menu is visible, the user can close the menu by touching the menu button on the screen again or by pushing the hardware back button. Selecting an item on the menu results in a completely different abstract state/screen of the app. Pressing the Option button leads the app to the configuration screen (s_2). The app also allows the user to navigate the file system (abstract state/screen s_0 and s_3). Note we intentionally made the root directory, the **/foo** directory, and the **/foo/foo** directory to have the same look to reduce the size of the example.

screen. Let us assume that there are no event handlers associated with the menu open and close events. The execution of the above action sequence will then generate the following trace:

$$s_0 \xrightarrow{Menu,\emptyset} s_1 \xrightarrow{Menu,\emptyset} s_0 \xrightarrow{Menu,\emptyset} s_1 \xrightarrow{Option,C_o} s_2.$$

where $\emptyset$ is the empty set and C_o denotes the test coverage generated when the app moves to the configuration screen. In this trace, the sub trace $s_0 \xrightarrow{Menu,\emptyset} s_1 \xrightarrow{Menu,\emptyset} s_0$ forms a loop since it begins and ends with the same abstract state or screen.[2] The coverage of this loop only contains the states s_1 and s_0. The coverage is a subset of the coverage of the rest of the trace, which is $C_o \cup \{s_0, s_1, s_2\}$. Thus the loop does not add any extra coverage than what is already achieved by the rest of the trace. This makes sense because the loop merely

[2]Note that the trace contains one more loop: $s_1 \xrightarrow{Menu,\emptyset} s_0 \xrightarrow{Menu,\emptyset} s_1$. We are going to ignore it for now.

opens and closes the pop-up menu. After removing the loop from the trace, if the trace is replayable, we can replace the original trace with the modified trace. Removing the loop gives us the following shorter replayable trace:

$$s_0 \xrightarrow{Menu,\emptyset} s_1 \xrightarrow{Option,C_o} s_2$$

. ▲

Example 3.2. (A non-redundant loop)

A loop is non-redundant when the loop provides coverage that cannot be achieved by the rest of the trace(s). Let us assume that the app now has event handlers attached to the menu open and close events. Re-executing the same sequence of actions from Example 3.1 will generate the following slightly different trace:

$$s_0 \xrightarrow{Menu,C_p} s_1 \xrightarrow{Menu,C_c} s_0 \xrightarrow{Menu,C_p} s_1 \xrightarrow{Option,C_o} s_2,$$

where C_p and C_c denote the coverage generated by executing the menu open and close event handlers, respectively. In this modified trace, the loop contributes the test coverage C_c, which cannot be achieved by any other transition in the trace. Therefore, the loop should not be removed from the trace. ▲

Example 3.3. (Another non-redundant loop)

A loop can be non-redundant if the removal of the loop makes the trace non-replayable, even if it does not achieve any new coverage. Note that if we use the concrete state of the app and its environment instead of a screen abstraction, a trace will be replayable if we remove a loop. However, since each s_i denotes an abstract state, the start and end states of a loop may not correspond to the same concrete state. Therefore, the trace may not be replayable after the removal of a loop. Let us illustrate this with an example. This time, we are going to navigate the file system to reach the folder containing pictures (i.e. reach the state s_3). This task can be done by simply touching the `foo` folder three times. The execution of the sequence of actions will generate the following trace:

$$s_0 \xrightarrow{foo,C_{f1}} s_0 \xrightarrow{foo,C_{f1}} s_0 \xrightarrow{foo,C_{f2}} s_3,$$

where C_{f1} denotes the test coverage generated when opening a folder only containing sub-folders, and C_{f2} denotes the test coverage generated when opening a folder only containing files. The trace has three loops (the first transition, the second transition, and the sub trace containing the first two transitions). The third loop cannot be removed because removing it will reduce the coverage of the trace. The first and seconds loops, however, look identical and one may think that one of the two loops can be removed from the trace. However, removing one of these two loops will make the trace non-replayable because touching the `foo` folder twice leads the app to the screen showing the contents of `/foo/foo` folder and we will miss C_{f2} and s_3. The trace obtained after removing one of the loops is non-replayable

because our state abstraction is coarse-grained which maps three distinct app states to s_0. However, if we do not use the abstraction, we will have unbounded number of abstract states which will make both automated test generation and test minimization fail. This example shows that a loop is non-redundant if its removal makes the trace non-replayable. Note that whether removing a loop will have an impact on the rest of a trace can only be determined by trying to replay the modified trace. ▲

Redundant Trace Fragments

While analyzing traces in test suites of several apps generated by an automated testing tool, we observed that many traces share common sub traces. If we execute these traces, the common sub traces get executed multiple times, i.e. once for each trace without contributing any new coverage over what is achieved when a common sub trace is executed for the first time. We can avoid redundant execution of these common sub traces if we can somehow combine fragments of traces in a way so that we can avoid repetitions of common sub traces. Combining fragments of traces will also result in longer traces. Such longer traces will reduce the number of restarts, which are more expensive operations than triggering an action. We next describe using examples how common sub traces contribute to redundancy.

Example 3.4. (Redundant sub trace)
Consider following two sample traces.

$$s_0 \xrightarrow{a,C_1} s_1 \xrightarrow{b,C_2} s_2 \xrightarrow{c,C_3} s_3 \xrightarrow{d,C_4} \boxed{s_4} \qquad s_0 \xrightarrow{a,C_1} s_1 \xrightarrow{b,C_2} s_2 \xrightarrow{e,C_4} \boxed{s_4} \xrightarrow{f,C_5} s_5$$

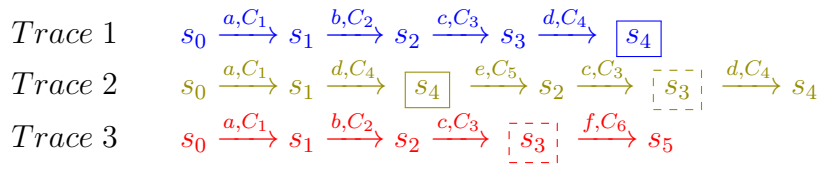
For simplicity of exposition, let us assume that each s_i is a unique screen, and the coverage sets $C_1, C_2, \ldots, C_5$ are mutually exclusive. Note that the two traces have a common prefix $s_0 \xrightarrow{a,C_1} s_1 \xrightarrow{b,C_2} s_2$. Note that the two traces have a common state s_4 in addition to the common prefix. This means that executing action f at the end of the first trace may enable us to replay the last transition of the second trace, resulting in the following trace:

$$s_0 \xrightarrow{a,C_1} s_1 \xrightarrow{b,C_2} s_2 \xrightarrow{c,C_3} s_3 \xrightarrow{d,C_4} \boxed{s_4} \xrightarrow{f,C_5} s_5.$$

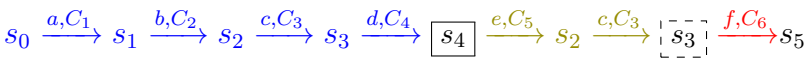
We use the color black to denote a state where two trace fragments were combined. The spliced trace may or may not be replayable because s_4 is an abstract state. Therefore, we need to check its replayability. If the spliced trace is replayable, we can use the spliced trace to test the app instead of the original two traces. We got rid of the redundant common prefix from the second trace. The spliced trace avoids three actions by skipping the uninteresting prefix of the second trace and one restart operation by combining two traces into one. ▲

The above example demonstrated a simple case involving two traces and a common prefix. Let us consider a more complicated example to see the potential of the idea of splicing trace fragments.

Example 3.5. (Splicing three traces)
Consider the following three artificially crafted traces:



Note the first and second traces have two common sub traces: $s_0 \xrightarrow{a,C_1} s_1$ and $s_2 \xrightarrow{c,C_3} s_3 \xrightarrow{d,C_4} s_4$. Similarly, the first and third traces have the common prefix $s_0 \xrightarrow{a,C_1} s_1 \xrightarrow{b,C_2} s_2 \xrightarrow{c,C_3} s_3$. By combining fragments of the traces, we can create the following trace:



The spliced trace is constructed by appending sub trace $s_4 \xrightarrow{e,C_5} s_2 \xrightarrow{c,C_3} s_3$ to the first trace, and then by appending the sub trace $s_3 \xrightarrow{f,C_6} s_5$ to the resulting trace. The new trace gets rid of six actions and two restart operations from the original traces. Note that the spliced trace still contains two copies of the sub trace $s_2 \xrightarrow{c,C_3} s_3$, which we could not get rid of. If the spliced trace is replayable, it can replace the original traces in the test suite. The running time of the spliced trace will be approximately half of the original traces, while providing the same coverage. This example shows that one could aggressively combine fragments from multiple traces while getting rid of redundant fragments (including redundant prefixes). Such aggressive splicing helps us to achieve more reduction. However, in practice, we also found that a trace composed of a large number of fragments from different traces tends to be non-replayable. Therefore, in our technique we limit the number of different trace fragments that we can combine to a small bound, which is three in our implementation. ▲

Our Approach

State abstraction. In our discussion so far, we assumed we have a suitable state abstraction that enables us to cluster similar looking screens of the app. The performance of our technique for test minimization depends heavily on our choice of state abstraction. If we choose a fine-grained abstraction, then our technique runs faster, but misses a lot of opportunities for reduction. For example, the algorithm will not identify and remove some loops. On the other hand, if we pick a coarse-grained abstraction, a lot of traces that we construct in our technique become non-replayable. Therefore, our technique spends a lot of time in checking replayability of various traces, but we get a bigger reduction by eliminating more loops compared to the case of using a coarse-grained abstraction.

We observed that a human tester can easily identify screens that are similar by analyzing what is visible on the screen. In our technique we needed an abstraction that will tell that two app screens are the same with respect to the abstraction if and only if a human tester finds the two screens visually identical. After analyzing several apps, we picked a suitable state abstraction that abstracts the screen of an app using information from the GUI component tree. Intuitively, a GUI component tree represents what is visible on the

screen. Therefore, by carefully analyzing a GUI component tree, we should be able to identify high level information that a human tester would use to infer if two screens are similar. Our abstraction uses a set of enabled actions from a GUI component tree (e.g., clicking a `CheckBox` type component) and augments each action with some details captured from the GUI component tree (e.g., the checkbox is the third child of the root component, and the box is checked). We found this abstraction works well in practice. The details of the abstraction are described in Chapter 4.

Removing redundancies. We propose a two-phase algorithm to remove redundancies from a GUI test suite. The first phase removes redundant traces and redundant loops greedily. It first removes redundant traces by greedily selecting traces in a way that each selected trace contributes new coverage to the coverage of the set of already selected traces. The non-selected traces are then redundant and are removed from the test suite. It then removes redundant loops from each remaining trace. In order to remove redundant loops in a trace, the algorithm creates the set of all traces obtained from the trace by removing zero or more loops. It then selects a trace from the set that does not decrease cumulative coverage, lowers cost of the trace maximally, and is replayable. Such a trace replaces the original trace in the test suite.

The second phase removes redundant trace fragments as much as possible. For that it constructs a new set of traces by combining fragments of the traces in the set computed by the first phase. When splicing trace fragments, we found it useful to limit the number of fragments that each spliced trace can have to a small number, which is three in our case, because a trace composed of many fragments tends to be non-replayable in practice. Thus, the second phase of the algorithm first creates the set of candidate traces composed of a bounded number of trace fragments. It then constructs a new test suite by greedily selecting traces from the set of candidate traces.

In both phases, whenever our algorithm generates a new trace, it checks whether the trace is replayable or not by executing it a few times. This helps the algorithm to avoid adding a non-replayable trace to the resulting regression test suite. If the algorithm finds a trace to be non-replayable, it identifies the shortest prefix of the trace that is non-replayable and saves it. In the future, if the algorithm finds that a new trace has a prefix that it has saved, then it can safely infer that the trace is non-replayable and discard it. This optimization helps the algorithm to aggressively discard some non-replayable traces without executing them multiple times. We describe the reduction algorithm formally in the next section.

3.2 Algorithm

Redundant Loop and Trace Elimination to Reduce Number of Transitions

In order to construct a minimal set of traces, we only retain the traces from T_s whose cumulative coverage is same as the coverage of T_s. We then remove as many loops from the remaining traces as possible while maintaining the same cumulative coverage and the replayability of the traces. This results in a set of traces T_r whose cost is much lower than the cost of T_s. During the removal of loops, our algorithm discovers that certain trace prefixes are not replayable. We speedup the loop elimination phase by pruning out the traces whose prefix matches the non-replayable prefixes. We next describe this algorithm formally.

Given a trace τ, we say that a sub-trace of τ is a *loop* if it begins and ends in the same state. For example, if in the trace $\tau = s_0 \xrightarrow{a_1,C_1} s_1 \xrightarrow{a_2,C_2} \ldots \xrightarrow{a_n,C_n} s_n$ there exists two states s_i and s_j such that $i \neq j$ and $s_i = s_j$, then the sub-trace $\ell = s_i \xrightarrow{a_{i+1},C_{i+1}} \ldots \xrightarrow{a_j,C_j} s_j$ is a loop. If we remove the loop from τ, we get a shorter trace $\tau_\ell = s_0 \xrightarrow{a_1,C_1} \ldots \xrightarrow{a_i,C_i} s_i \xrightarrow{a_{j+1},C_{j+1}} \ldots \xrightarrow{a_n,C_n} s_n$. A trace obtained after eliminating one or more loops from τ may not be replayable anymore. Therefore, anytime our algorithm removes a loop from a trace, we need to check if the resulting trace is still replayable. Let $L(\tau)$ be the set of all traces obtained by removing different combinations of zero or more loops from τ. Note that $L(\tau)$ contains τ.

The pseudocode of the algorithm is shown in Algorithm 4. The algorithm uses the function Replay, which takes a trace τ and returns τ if the trace is replayable. When τ is not replayable, there are two possible reasons. First, it is possible that re-executing the actions of τ gives the expected sequence of screens, but fails to provide the expected test coverage. In such a case, Replay returns the new trace with the correct test coverage information. Another possibility is that the re-execution gives a different screen at some point. If this is the case, the Replay function returns the shortest prefix of τ that is not replayable. In both cases, the check $\tau = \text{Replay}(\tau)$ indicates whether τ is replayable. The algorithm also uses the CheckScreens function to determine whether two traces have the same sequence of screens. In the first part of the algorithm, we remove all redundant traces. To do this, we create an empty set T to store the non-redundant traces. The algorithm goes over each trace τ in T_s. If $C(\tau)$ has coverage that is not already present in $C(T)$, then τ is not redundant and we add τ to T. After going over all traces in T_s, T will contain non-redundant traces of T_s such that $C(T) = C(T_s)$.

In the second part, the algorithm performs redundant loop elimination. It maintains a set of reduced traces T_r, which is initialized to the empty set. The algorithm goes over each trace τ in T, then over each trace τ' in $L(\tau)$ (the set of all traces obtained from τ by removing zero or more loops) in the order of increasing cost. The algorithm also computes T_{syn}, the set of all synthesized traces in $L(\tau)$. If $C(\tau') \cup C(T_r) = C(\tau) \cup C(T_r)$ and τ' is replayable—that is, if $\tau' = \text{Replay}(\tau')$—the algorithm adds τ' to T_r and stops processing elements of $L(\tau)$. This indicates that the algorithm has computed a trace possibly shorter than τ. However, if

τ' is not replayable there can be two cases. If candidate trace τ' is a synthesized trace and re-executing the sequence of actions of τ' gives the same sequence of screens, this indicates that the test coverage information of τ' was incorrect because τ' is synthesized. In such a case, the algorithm adds the result of the REPLAY function call to T, in order to give another chance to the candidate trace with the correct test coverage information. Otherwise, τ' is discarded and any trace in $L(\tau)$ having REPLAY(τ') as a prefix is removed from the set $L(\tau)$ because all such traces will also be non-replayable. This reduces the number of the traces that we must process from the set $L(\tau)$, and thus helps to optimize the running time of the algorithm. Note that during the processing of the traces in $L(\tau)$, we will end up adding τ to T_r if none of the loops in τ can be eliminated without reducing coverage or without the resultant trace non-replayable. The algorithm always terminates because the two outer loops iterate exactly $|T_s|$ times and $|T|$ times, respectively, and the inner loop (lines 14 - 26) can use at most $2 * |L(\tau)| - 1$ candidate traces.

Practical concerns. The algorithm relies on a robust implementation of REPLAY(τ). However, in practice it is not easy to have a precise implementation of REPLAY(τ) that will guarantee that REPLAY(τ) returns τ if and only if τ is replayable. Such an implementation would require us to track the entire state of the app including the state of any internet server it might be interacting with. Moreover, if we make the implementation of REPLAY(τ) too precise, in many acceptable cases it will report that τ is not replayable. In our tool, we make a practical trade-off where we re-execute the trace τ a few times, which is ten in our experiments. If in all the executions we find that τ is replayable, REPLAY(τ) returns τ. If executing the action sequence of τ generates trace τ' that has the expected sequence of screens but does not to provide the expected test coverage, REPLAY(τ) returns τ'. Otherwise, REPLAY(τ) reports the shortest prefix of τ that is non-replayable over all ten re-executions.

The algorithm also needs to compute $L(\tau)$, i.e. the set of traces obtained from τ by removing 0 or more loops. Our implementation does not compute the set $L(\tau)$ ahead of time. Rather it performs a depth-first traversal of the trace τ to enumerate the traces in $L(t)$ one-by-one from the shortest to the longest one.

Finally, the result of the first phase of the algorithm will change depending on the order of selecting elements from T_s (line 3) and T (line 10). Our implementation uses queues to store test cases, which guarantees that test cases are always handled in first-come first-serve order.

Trace Splicing

While analyzing traces in the set T_r, i.e. the traces generated by loop and trace elimination, we noticed traces often share common sub traces. Therefore, if we can combine traces in a way so that we can avoid common sub traces as much as possible, we will generate longer traces, which will have a couple of advantages: longer traces will avoid expensive restarts and they will avoid execution of redundant sub traces. However, we also found that the more traces we combine, the more likely we will get traces that are non-replayable. Specifically, we

Algorithm 4 Eliminate redundant traces and loops

```
1: procedure EliminateRedundantTracesAndLoops(T_s)                ▷ T_s is the input trace set.
2:     T ← ∅                                                      ▷ Part 1: Eliminate redundant traces.
3:     for τ ∈ T_s do                                             ▷ Loop over the input trace set.
4:         if C(τ) ⊈ C(T) then
5:             T ← T ∪ {τ}                                        ▷ Collect τ if it has a unique coverage.
6:         end if
7:     end for
8:
9:     T_r ← ∅                                                    ▷ Part 2: Eliminate redundant loops.
10:    for τ ∈ T do                                               ▷ Loop over the filtered trace set T.
11:        T_L ← L(τ)                                             ▷ L(τ) is a set of traces obtained by removing zero or more loops from τ.
12:        T_syn ← T_L \ {τ}                                      ▷ T_syn is a set of synthesized candidate traces.
13:        while T_L ≠ ∅ do
14:            τ' ← argmin_{τ∈T_L} |τ|                            ▷ Get the shortest element of L(τ).
15:            T_L ← T_L \ {τ'}
16:            if C(τ') ∪ C(T_r) = C(τ) ∪ C(T_r) then
17:                if τ' = Replay(τ') then
18:                    T_r ← T_r ∪ {τ'}                           ▷ τ' is replayable and keeping the coverage of τ.
19:                    break the inner loop
20:                else if τ' ∈ T_syn ∧ CheckScreens(τ', Replay(τ')) then   ▷ τ is a synthesized trace.
21:                    T_k ← T_k ∪ {Replay(τ')}                   ▷ Give one more chance with the corrected test coverage information.
22:                else                                           ▷ Replay(τ') is the shortest prefix of τ' that is not replayable.
23:                    T_L ← {τ ∈ T_L | Replay(τ') is not a prefix of τ}   ▷ Remove non-replayable traces from T_L.
24:                end if
25:            end if
26:        end while
27:    end for
28:    return T_r                                                 ▷ Return the reduced trace set T_r.
29: end procedure
```

found experimentally that if we combine three or fewer trace fragments, we can get longer traces that avoid restarts and redundant execution while being replayable. Based on these observations, we devised the second part of our minimization algorithm where we combine fragments from different traces to create longer replayable traces.

A *trace fragment* is a contiguous portion of a trace obtained by removing a prefix and a suffix of the trace. For example, if $\tau = s_0 \xrightarrow{a_1,C_1} s_1 \xrightarrow{a_2,C_2} \ldots \xrightarrow{a_n,C_n} s_n$ is a trace, then for any $i, j \in [0, n]$ where $i \leq j$, $s_i \xrightarrow{a_{i+1},C_{i+1}} \ldots \xrightarrow{a_j,C_j} s_j$ is a *fragment* of the trace τ. A set of trace fragments $\tau_1, \tau_2, \ldots, \tau_m$ can be combined to form the trace $\tau_1\tau_2 \ldots \tau_m$ if τ_1 begins with the state s_0 and for all $i \in [2, m]$, the end state of τ_{i-1} is the same as the first state in τ_i. Given a set of traces T_r, let T_k be the set of all traces obtained by combining at most k trace fragments obtained from the traces in T_r.

The pseudocode of the algorithm is shown in Algorithm 5. The algorithm first constructs the set T_k from the set T_r. The algorithm also computes T_{syn}, the set of all synthesized traces in T_k, and initializes the final set of minimized traces T_m to the empty set. The algorithm then performs the following in a loop: First, it finds a trace τ in T_k such that τ results in the maximal increase in coverage over the coverage of T_m—that is, τ maximizes $|C(\tau) \setminus C(T_m)|$. If no such trace is found in T_k, the algorithm returns T_m. Otherwise, the algorithm removes τ from T_k and checks whether it is replayable. If τ is replayable, the algorithm adds τ to T_m. When τ is not replayable, there can be two cases. If candidate trace τ is a synthesized trace

Algorithm 5 Bounded splicing

```
1: procedure BoundedSplicing(T_r, k)          ▷ T_r is the input trace set. k bounds the number of trace fragments.
2:   T_k ← {τ | τ is a trace composed of at most k fragments of traces in T_r}   ▷ T_k is a set of candidate traces.
3:   T_syn ← T_k \ T_r                        ▷ T_syn is a set of synthesized candidate traces.
4:   T_m ← ∅                                  ▷ We use T_m to collect selected traces.
5:   while ∃τ ∈ T_k.C(τ) \ C(T_m) ≠ ∅ do      ▷ Iterate until no trace in T_k can increase coverage.
6:     τ ← argmax_{τ∈T_k} |C(τ) \ C(T_m)|     ▷ τ is the candidate maximizing the increase in coverage.
7:     T_k ← T_k \ {τ}
8:     if τ = Replay(τ) then                  ▷ Select the candidate if it is replayable.
9:       T_m ← {τ}
10:    else if τ ∈ T_syn ∧ CheckScreens(τ, Replay(τ)) then   ▷ τ is a synthesized trace.
11:      T_k ← T_k ∪ {Replay(τ)}   ▷ Give one more chance with the corrected test coverage information.
12:    else                        ▷ Replay(t) is the shortest prefix of t that is not replayable.
13:      T_k ← {τ ∈ T_k | Replay(τ) is not a prefix of τ }   ▷ Skip the candidate and remove non-replayable candidate traces.
14:    end if
15:  end while
16:  return T_m                              ▷ Return the result of splicing.
17: end procedure
```

and Replay(τ) and τ have the same sequence of screens, the algorithm adds the result of the Replay function call to T_r, in order to give another chance to the candidate trace with the correct test coverage information. Otherwise, all traces in T_k with Replay(τ) as a prefix are removed from T_k. This step speeds up the search for optimal τ in future iterations. The loop is then repeated.

This algorithm terminates and computes a T_m such that $C(T_m) = C(T_r)$. The algorithm terminates because in each iteration we increase the coverage of T_m and the coverage of T_m cannot be increased beyond the coverage of T_r, which is the same as the coverage of T_k. Moreover, the algorithm ensures that $C(T_m) = C(T_r)$ because for any finite k, T_k contains the traces in T_r. Therefore, in the worst case if none of the traces obtained by combining two or more trace fragments from different traces are replayable, we will end up adding all the traces in T_r to T_m, which will ensure that $C(T_m) = C(T_r)$.

Computing T_k. The above description of the splicing algorithm uses a declarative specification to describe the trace set T_k. We next describe an algorithm to compute the set efficiently. For any set of traces, we can construct a labeled transition system composed of the transitions of the traces in the set. Formally, if T_r is a set of traces, we can construct a labeled transition system $Q_{T_r} = (S, s_0, L, A, \mathcal{C}, \delta)$, where

- S is the set of all states in T_r,
- s_0 is the initial state of the app,
- $L \subseteq \mathcal{N} \times \mathcal{N}$ is a set of labels,
- A is the set of all actions in T_r,
- $\mathcal{C}$ is the set of coverage sets in T_r, and

- δ is a set of labeled transitions of the form $s_{i-1} \xrightarrow[l]{a_i, C_i} s_i$, which is defined as

$$\delta = \{s_{i-1} \xrightarrow[(j,i)]{a_i, C_i} s_i \mid \exists \tau_j \in T_r \text{ such that } s_{i-1} \xrightarrow{a_i, C_i} s_i \text{ is the } i^{\text{th}} \text{ transition in } \tau_j\}.$$

  Informally, an element of δ is a transition from a trace in T_r, augmented with a pair of indices denoting the trace to which the transition belongs and the position of the transition in the trace.

Note that for any trace in T_r, there is a path in the labeled transition system Q_{T_r}. Moreover, any path in Q_{T_r} represents a trace that could be obtained by combining trace fragments from T_r. We can check if a path from Q_{T_r} belongs to T_k by analyzing the labels of the path as follows. Two consecutive transitions with labels (j_1, i_1) and (j_2, i_2) in a path constitute a *switch* if either $j_1 \neq j_2$ or $i_1 + 1 \neq i_2$. A path in Q_{T_r} belongs to T_k if the number of switches in the path is less than k. The algorithm to construct T_k enumerates the paths in Q_{T_r} using depth-first search and discards a path as soon as the number of switches along the path reaches k. The algorithm terminates because k is finite and the number of trace fragments is bounded.

Chapter 4

Implementation

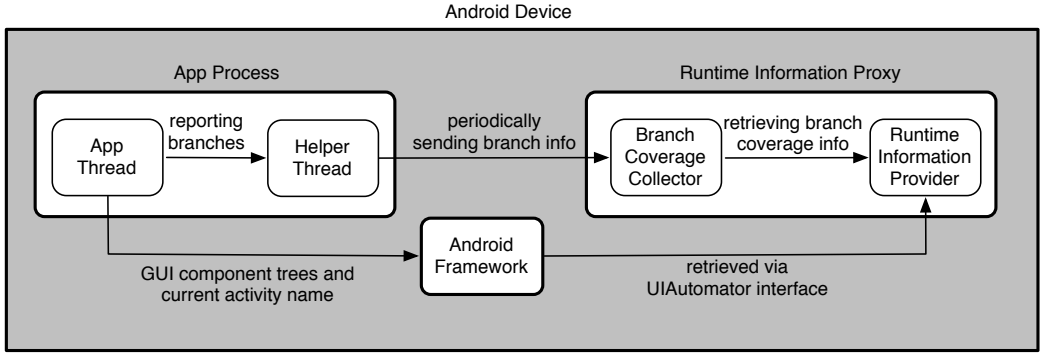
This chapter describes the implementation of our automated testing infrastructure for Android apps. We created a testing infrastructure that provides common building blocks for various testing algorithms, and implemented SwiftHand, $\mathcal{L}^*$, random, and DetReduce on top of this infrastructure. The initial testing infrastructure was built to implement and evaluate SwiftHand using *asmdex* [85], a Dalvik bytecode instrumentation library, and *chimpchat*, a low-level test automation library for Android. Later we reimplemented the infrastructure due to instability issues in the underlying libraries. The new implementation is built on top of *Soot* [110] and *UIAutomator* [3]. The infrastructure itself is written in Java. In this thesis, we use the reimplemented version of the infrastructure to evaluate both SwiftHand and DetReduce. We have made the latest version publicly available at `https://github.com/wtchoi/swifthand2` under the three-clause BSD license.

The rest of the chapter is organized as follows: Section 4.1 offers an overview of the design of the automated Android testing infrastructure. Section 4.2 , Section 4.3, and Section 4.4 explain how screens are abstracted, how Android app packages are instrumented, and what limitations remain in the current testing infrastructure implementation, respectively. We refer readers to our conference paper on SwiftHand [23] for details about implementing the older version of the testing infrastructure.

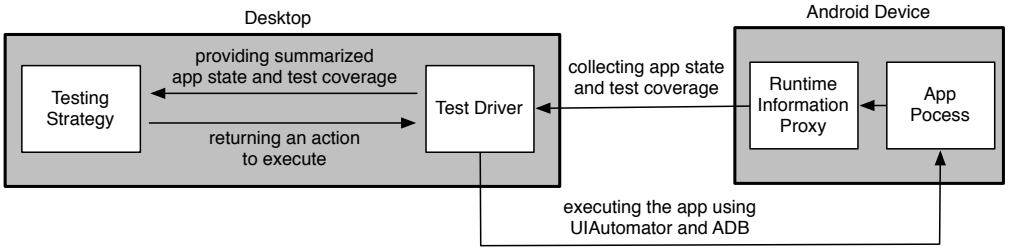
4.1 Design Overview

The testing infrastructure is composed of four major components: the instrumentation tool, runtime information proxy, test driver, and testing strategies. Figure 4.1a and Figure 4.1b show how the components work together during the test execution. Note that the APK instrumentation tool does not appear in the diagram, because it runs before starting a testing session.

- **APK instrumentation tool:** The APK instrumentation tool instruments an Android app to collect runtime information during its execution and to send this information to a runtime information proxy on the same device. Once the information is transmitted to



(a) Interaction between the instrumented app and runtime information proxy.



(b) Interaction between the test driver and rest of the components.

Figure 4.1: The structure of the testing infrastructure

the proxy, the instrumented app discards the information. Other than the existence of the runtime information proxy, an instrumented app is unaware of the rest of the testing infrastructure. In the current version of the testing infrastructure, instrumented apps collect only branch coverage information. Further details about the instrumentation tool are explained in Section 4.3.

- **Runtime information proxy:** A runtime information proxy is a process running on the same device as the target app. When a new testing session begins, the testing infrastructure first launches the runtime information proxy before initiating the actual testing. During the testing session, the proxy receives branch coverage information from the target. The proxy also collects other information—such as GUI component trees and whether the target app is in a stable state—using UIAutomator, a low-level Android GUI testing library. The proxy runs a simple server over TCP, to which a test driver can connect and retrieve the collected information. The proxy collects the following information from the app under test.
  - **GUI component trees:** GUI component trees are collected using the `UiSelector` interface of UIAutomator and the reflection API of Java. `UiSelector` retrieves information about a set of GUI components satisfying a given search condition. However, the information provided by `UiSelector` is limited—for example, it is

insufficient to determine the hierarchy among the GUI components. The runtime proxy overcomes this limitation by accessing private fields of `UiSelector` objects using the Java reflection API.

  - **Transition completion:** When a user input is sent to an Android app, a number of event handlers are executed and the screen content is modified. After sending a user input, a testing algorithm must wait until all event handler executions and screen modifications are complete. The proxy uses an API provided by UIAutomator to detect the completion of a state transition, the `waitForIdle` method of the `UIDevice` class. Once invoked, this method waits until the GUI of the currently running app enters an idle state. However, it cannot guarantee that all event handler executions are complete. Therefore, the proxy also checks whether the target app is sending any branch coverage information. If no branch coverage information has been recently sent from the app, the proxy concludes that there is no running event handler. The current implementation waits 100 milliseconds. By combining these two mechanisms, the proxy checks whether a state transition has been completed.
  - **App liveness:** The proxy also checks whether the target app is alive. An Android app is composed of a set of activities. Each activity implements a single application screen and its functionality. When an app is terminated, all activities associated with the app are stopped. Thus, the problem of checking app termination reduces to the problem of tracking the status of the app's activities. The proxy uses the `getCurrentActivityName` method of the `UIDevice` class provided by UIAutomator to identify the foreground activity—if the foreground activity does not belong to the app under test, the proxy concludes that the target app has been terminated.

- **Test driver:** The test driver is the main entry point of the testing infrastructure. Because of the computational demand, the test driver runs on a desktop or laptop machine and communicates with the target app and the proxy via ADB. At the beginning of a session the driver launches both the target app and the runtime information proxy on the target device. The driver also selects and initializes a testing strategy. Once all components are ready, the driver begins the testing loop, which performs several tasks. First, it combines pieces of runtime information gathered from the target app into a package and relays that package to a testing strategy. Most information is collected via the runtime information proxy, and the test driver adds two additional pieces of information to the package.

  - **Exceptions:** The test driver checks whether an uncaught exception is raised from the target app. Such exceptions can be checked by analyzing the Android system log messages, as uncaught exceptions are always logged and always tagged with the ``` ``AndroidRuntime'' ``` keyword. Thus, the driver retrieves the system log

messages by using the ADB `logcat` command, then searches exceptions from the logs.

- **Screen abstraction:** The test driver computes a screen abstraction and infers a set of enabled inputs from the current screen information (given as a GUI component tree), then adds it to the package. The details of the screen abstraction mechanism are explained in Section 4.2.

Second, once all necessary information is gathered and delivered to the testing strategy, the test driver then receives high-level commands (`Actions`), from the testing strategy and executes them by using a combination of underlying library functions. The test driver currently supports three types of actions:

- `Start`: The `Start` action is used to initiate the target app, and it is used without any parameter. If the app is not started, the test driver launches the app using ADB (via `shell am start` command). If the app is already running, the test driver does nothing.
- `Close`: The `Close` action is used to close the target app, and used without any parameter. If the app is running, the test driver first kills the app through (via `shell am force-stop` command), then removes app data. Because the Android file system is sandboxed, the data private to each app can be cleared with a simple ADB command (`shell pm clear`). The only exception is the contents of SD cards. Because removing an app or clearing the app's private data will not remove persistent data from SD cards, we must explicitly clear SD card contents. The test driver clears the contents of SD cards using ADB (via `shell rm` command). If the app is already terminated, the `Close` action only cleans the app data and the contents of SD cards. We implement the clean restart operation by executing the `Close` and `Start` actions in sequence.
- `Event`: The `Event` action triggers one of the enabled inputs of the current screen, taking an integer (the identifier of the input to be triggered) as a parameter. Upon receiving the `Event` action, the test driver executes the action using UIAutomator, which provides APIs to inject various kinds of user inputs to the target app, including touch, long touch, scroll, and key stroke. UIAutomator also supports replacing the contents of editable text boxes, such as *EditText*. If the target app is inactive, the test driver does nothing.

The test driver handles all actions in a synchronous manner—that is, after executing an action the test driver waits until the app transition caused by the action completes. The test driver determines whether the transition is completed based on the information sent from the runtime information proxy. Once the action is fully handled, the driver returns to the beginning of the loop.

- **Testing strategies:** Testing strategies, such as SwiftHand, $\mathcal{L}^*$, and DetReduce, guide automated testing sessions by communicating with the test driver. Test strate-

gies run on the same machine as the test driver. A single testing session can apply only a single strategy, but each testing session is free to choose any strategy. Testing strategies utilize the information provided by the test driver to make high-level decisions independent of low-level details. Each strategy is implemented as a module of several standardized API methods that the test driver uses to communicate with strategies. There are two key methods in the testing strategy API:

- `void reportCurrentAppState(AppState s, Coverage c)`
- `Action getNextAction()`

The test driver invokes `reportCurrentAppState` to report the state of the target app and the test coverage achieved by executing the most recent action. A testing strategy can use the received information to build a model or to determine the next action to perform. After giving the information to the strategy, the driver invokes `getNextAction` of the strategy to retrieve a further action to execute.

4.2 Screen Abstraction

Both SwiftHand and DetReduce use a suitable screen abstraction computed by the test driver to cluster app states. This screen abstraction is computed from a GUI component tree collected from an app via UIAutomator at runtime. A GUI component tree represents the hierarchical structure of GUI components on the screen of the app. The coordinate, size, and type information of each GUI component is also available. We observed that the following information is often sufficient to characterize a screen.

- Which GUI components are actionable? For example, *Checkbox*, *EditText*, and *TextButton* components are actionable in Android, while *TextBox* and *DecorationBar* components are not.
- Which attribute values of actionable GUI components are visible on the screen? For example, for a screen with a checkbox one could observe whether the box is checked. GUI components also contain several invisible attributes, such as `focused` and `focusable`, which we have found are not useful to characterize screens.

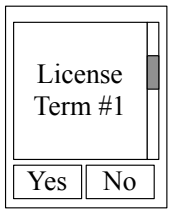
We use an abstraction to extract this information from a raw GUI component tree. The abstraction is computed by collecting a set of actionable GUI components (i.e., the components whose event handlers can be triggered by user actions) from a GUI component tree. The test driver determines whether a GUI component is actionable by checking two Boolean attributes of the components: *actionable* and *visible*. If both are true, the event handlers of the component can be triggered by user actions. Note that this method gives an over-approximate set of actionable GUI components, as some GUI components satisfying the above condition might not have attached event handlers. A potential concern is that such

over-approximation could make an automated testing algorithm less effective in achieving greater test coverage in a limited time; however, we found that injecting an action to a GUI component with no attached event handler keeps the app state unchanged. When executing such an idle transition, SwiftHand adds a self-loop to the current state of the app's model and avoids executing the loop afterwards. This learning capability, along with the aggressive state-merging, allows SwiftHand to execute only a small number of idle transitions. As such, this over-approximation does not pose a performance issue for SwiftHand. In contrast, an automated testing algorithm not based on a learning could in fact suffer from the over-approximation. DetReduce is independent of any over-approximation because it selects actions from the input test suite that it is minimizing.

After the abstraction collects the actionable GUI components, each collected component is augmented with the access path to the component from the root of the GUI component tree. For example, if a GUI component is the second child of the first child of the root GUI component, its access path will be $[0, 1]$. The access path of the root GUI component is [] (an empty list), because it does not have siblings and parents. We then remove all invisible attributes except type from each GUI component. The type attribute is necessary because it allows us to analyze what actions are meaningful for each GUI component. We also remove the coordinates and size from each GUI component even if they are visible attributes, because coordinates and size change easily and unpredictably when scrolling a list or editing a text box. Ignoring these attributes allows us to identify more logically identical screens. We have observed that this abstraction is appropriate for grouping screens that users might find similar.

Inferring Enabled Inputs from a Screen Abstraction

The abstraction also provides sufficient details to determine enabled user inputs. Given an abstract screen, the test driver can infer a representative set of enabled inputs by analyzing each element (a simplified GUI component) of the screen. For components that can be touched, such as a *Button* type component, a touch input corresponding to the component is added to the set of enabled inputs. If a GUI component is a subclass of *EditText*, then a user input capable of replacing the text contents of the GUI component is added to the set of enabled inputs. The infrastructure allows using any text. However, the current implementation of SwiftHand always uses a "random string" in order to restrict the app state space to test. Note that generating string values for testing is an independent research topic, and considering both GUI actions and string values simultaneously is an open problem we do not aim to solve in this thesis. For scrollable components, scroll inputs are added to the set of enabled inputs. Inputs corresponding to pressing the **Back** and **Menu** buttons are always added to the enabled input set.



(a) EULA screen

DecorView(1)
+ *LinearLayout*(2)
| + *ScrollableLayout*(3)
| | + *TextBox*(4)
| | *LinearLayout*(5)
| | + *Button*[Yes](6)
| | | *Button*[No](7)

(b) GUI component tree

{
ScrollableLayout(3),
Button[*Yes*](6),
Button[*No*](7)
}

(c) Abstract screen

{
ScrollDown(3),
ScrollUp(3)
Touch(6),
Touch(7),
Menu,
Back
}

(d) Enabled inputs

Figure 4.2: Screen abstraction and enabled input inference example

An Example

Figure 4.2 shows an EULA screen from the *Sanity* example, its simplified GUI component tree, the corresponding screen abstraction, and the corresponding enabled inputs. In the GUI component tree (Figure 4.2(b)), each component is described with its type and identifier. Note that here we use identifiers for ease of exposition. In the real implementation, access paths are used instead. For example, the identifier 5 of the second *LinearLayout* type component will be replaced by the access path $[0, 1]$ in the real implementation, which indicates that this is the second child of the first child of the root (the *DecorView* component). Among the components in the GUI component tree, two *Button* type components with ID 6 and ID 7 can be touched, and the *ScrollableLayout* type component can be scrolled. Therefore, we add these components to the screen abstraction (Figure 4.2(c)). From the screen abstraction, we can easily deduce the set of enabled inputs by adding **ScrollDown** and **ScrollUp** for the *ScrollableLayout*(3) component, and **Touch** actions for the two *Button* type components. We also add actions corresponding to pressing the **Back** and **Menu** buttons. Figure 4.2(d) shows the resulting set of enabled inputs.

4.3 APK Instrumentation Tool

Android apps are distributed as a single APK package file. To perform instrumentation, the testing infrastructure first extracts a program binary from a package file and injects the modified binary into the package file.

An APK file is a zip archive with a predefined directory structure. The archive contains a `class.dex` file, a `Manifest.xml` file, a `META-INF` directory, and other resource files. The `class.dex` file contains *Dalvik* byte code to be executed, the `Manifest.xml` file is a binary encoded xml file describing various parameters such as main activity and privilege setting, and the `META-INF` directory includes the app signature.

Figure 4.3 shows the flow of the app-modification process. First, the app package is unpacked. Then the `class.dex` and `Manifest.xml` files are instrumented. We instrument the dex file using *Soot* [110], a Java static analysis and instrumentation toolkit, to collect

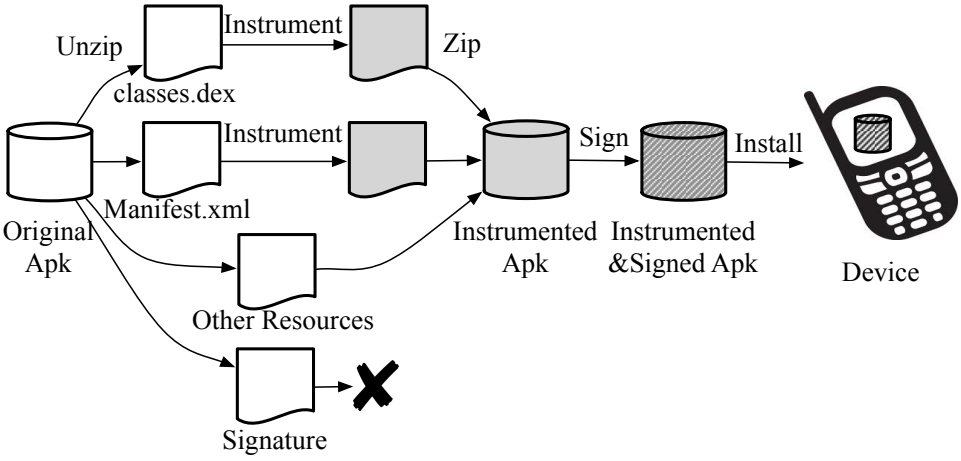


Figure 4.3: Flow of the App modification process

branch coverage information at runtime. The instrumentation makes two changes to the dex file.

- First, it injects the bytecode of a pre-compiled library into the dex file. The library provides a method that logs the execution of various branches in an app, taking the unique identifier of a branch as an argument. When invoked, the method records the identifier of the reported branch. It also launches a helper thread that periodically (every 50 milliseconds) sends the recorded branch information to the runtime information proxy if the thread is not yet initiated.
- Second, it adds a static method invoke instruction (**invokestatic**) that calls one of the methods of the injected library to the head of every basic block of every method. The instrumentation tool assigns a unique identifier to each basic block, and provides the identifier as the argument to the injected method invoke statement.

The xml file is modified using *axml* [4], an xml-encoded binary manipulation library. The `Manifest.xml` file must be modified because the modified app requires network access privilege to communicate with the runtime information proxy. The `META-INF` directory is removed as the original signature conflicts with the modified package. After modification, the app is repacked, signed, and delivered to experiment devices. For testing purposes, the key for the re-signed signature does not need to be identical to the key for the original signature.

4.4 Limitations

We next discuss the limitations of the testing infrastructure. First, the current implementation does not support apps whose main entry routine is native. Several games fall under

this category, and these are excluded from our benchmark suite. The testing infrastructure also cannot collect branch coverage information from apps equipped with a tamper-detection mechanism, such as LVL [1] or DexGuard [2], because such apps cannot be modified. These apps can be immediately identified because they do not start properly with binary instrumentation. We have also found that several apps from large companies, such as Microsoft and Google, are tamper-resistant. Although this restricts the applicability of the testing infrastructure, we did not find this limitation to be a serious problem while we were procuring apps to test.

Second, the current implementation works correctly only on devices running Android 4.3 or higher versions. This is a requirement of the UIAutomator library. However, this limitation does not restrict the applicability of the testing infrastructure in practice because more than 90% of activated Android devices run Android 4.3 or higher version. [3]

Finally, the current testing infrastructure does not support several features of the Android framework. For example, it ignores the contents of WebView components due to limitations of the UIAutomator library. The current implementation also does not support inter-app communication because of the implementation overhead. Although these two limitations restrict the maximum test coverage that can be achieved by an automated testing tool built on top of the infrastructure, they did not restrict the apps we could test.

[1] `https://android-developers.googleblog.com/2010/09/securing-android-lvl-applications.html`

[2] `https://www.guardsquare.com/en/dexguard`

[3] `https://developer.android.com/about/dashboards/index.html`

Chapter 5

Evaluation

This chapter provides an evaluation of the SwiftHand and DetReduce algorithms using several real-world Android apps. In Section 5.1 we briefly introduce the set of benchmark apps and the experimental environment. In Section 5.2 we evaluate the SwiftHand algorithm, showing that SwiftHand can achieve significantly better coverage than traditional random testing and $\mathcal{L}^*$-based testing in a given time budget. SwiftHand also reaches peak coverage faster than both random and $\mathcal{L}^*$-based testing. In Section 5.3 we evaluate the DetReduce algorithm, finding that DetReduce reduces a test suite by an average factor of 16.9 in size and 14.7 in running time with equivalent test coverage. We also find that given a test suite generated by running SwiftHand for eight hours, DetReduce can reduce such a test suite in 10.6 hours on average.

5.1 Experimental Setup

Benchmark Apps

We applied SwiftHand and DetReduce to eighteen free apps downloaded from the Google Play store [35] and F-Droid store [95]. Table 5.1 lists these apps along with their package name, the store from which that app was downloaded, the type of the app, and the number of branches (#Br) in the app (which offers a rough estimate of the size of the app.) Since the apps were downloaded directly from app stores, we have access to only their bytecode. Thirteen of these apps were used for experimental evaluation in previous research projects [24, 32, 125]; other apps, which we mark with asterisks, are newly selected. We avoided selecting apps that are highly non-deterministic or that cannot be restarted cleanly. For example, the Amazon online store app is highly non-deterministic because its main screen recommends a different set of items every time the app is launched. A mail app cannot be restarted cleanly because it depends on data stored on a third-party server, which we cannot reset easily. As such, we cannot automatically generate replayable test suites for such apps. We also excluded apps for which SwiftHand saturates the test coverage in less

Name	Package name	Market	Type	#Br
acar	com.zonewalker.acar	play	car management	20380
amemo	org.liberty.android.fantastischmemo	play	flashcard	6394
amoney*	com.kpmoney.android	play	finance	28141
astrid	org.tasks	play	task manager	16844
cnote*	com.socialnmobile.dictapps.notepad.color.note	fdorid	note	14524
dmoney	com.bottleworks.dailymoney	play	finance	5099
emobile	org.epstudios.epmobile	play	fitness tracker	3201
explore	com.speedsoftware.explorer	play	file system	54302
mileage	com.evancharlton.mileage	fdroid	car management	7728
mnote	jp.gr.java_conf.hatalab.mnv	fdroid	text editor	1959
monefy	com.monefy.app.lite	play	finance	22615
sanity	cri.sanity	play	device manager	4610
tippy	net.mandaria.tippytipper	fdroid	tip calculator	5243
todo*	com.splendapps.splendo	play	task manager	11858
ttable*	com.gabrielittner.timetable	play	scheduler	11858
vlc*	org.videolan.vlc	play	media player	14410
whohas	de.freewarepoint.whohasmystuff	fdroid	inventory	369
xmp	org.helllabs.android.xmp	play	media player	5855

Table 5.1: Benchmark apps. #Br means the number of branches in the apps.

than an hour. Note that adding such apps would only improve experimental results because most traces in test suites for such apps are redundant.

Experiment Environment

We used five smartphones (Motorola XT1565) running Android 6.1 to run the benchmark apps. We controlled the execution of the smartphones using a Linux machine running Ubuntu 16.04 equipped with an Intel E3-1225v3 CPU (four cores) and 32Gb RAM. We did not use more recent versions of Android (such as 7.X) because our smartphones do not support them.

5.2 Evaluation of SwiftHand

In this section, we evaluate SwiftHand on the apps shown in Table 5.1. We first compare SwiftHand with random testing and $\mathcal{L}^*$-based testing, focusing on the relative performance of the SwiftHand algorithm when all three algorithms inject events at the same speed. We then discuss the results and identify shortcomings in SwiftHand.

We use four hours as the testing budget per app for every strategy, implementing all three strategies on top of the testing infrastructure described in Chapter 4. In random testing, we restart an app with probability 0.1 and select an input from a set of enabled inputs uniformly at random. We are going to name the random testing algorithm as Random.

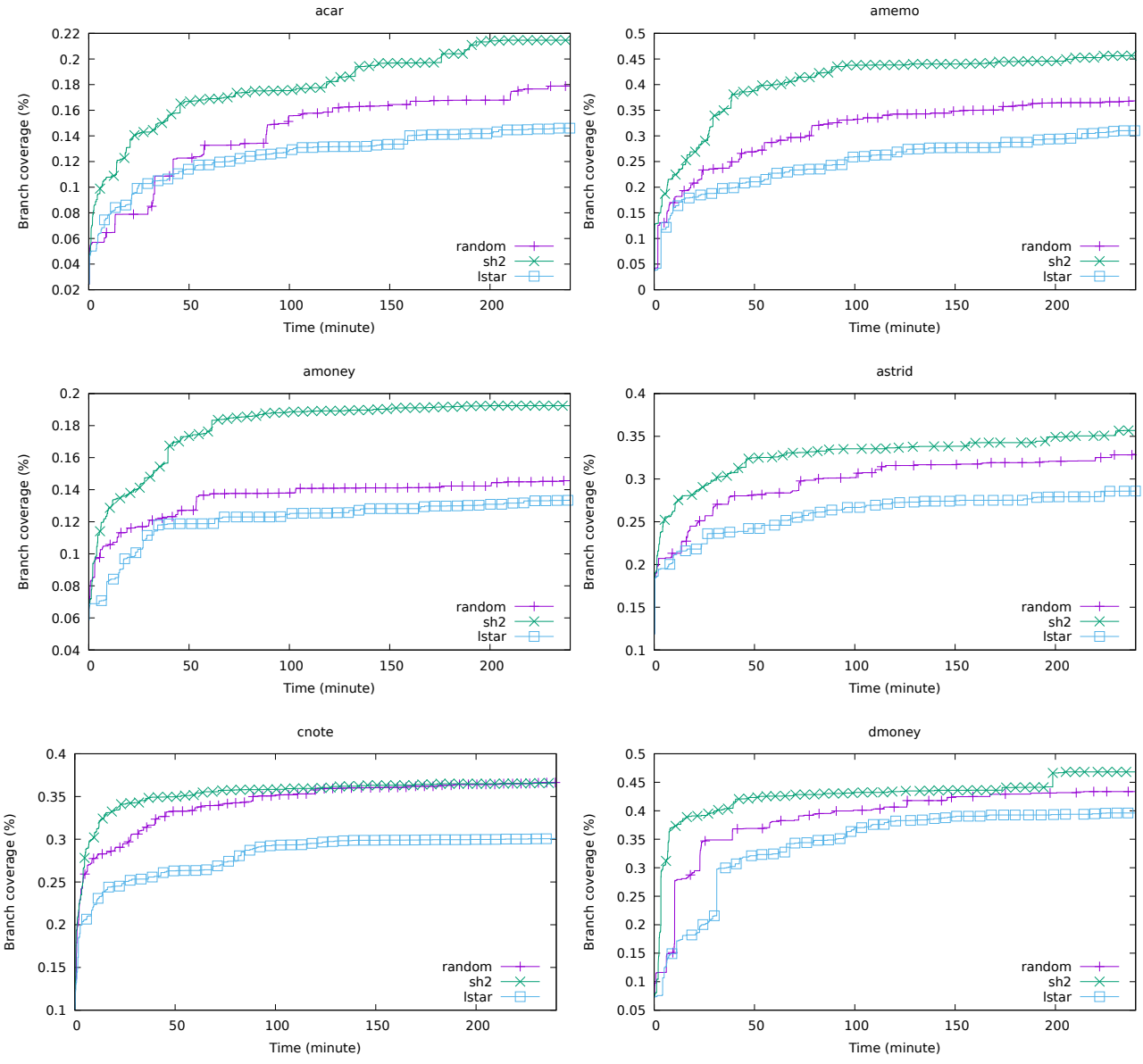


Figure 5.1: Comparison of the branch coverage of SwiftHand, Random and $\mathcal{L}^*$, #1

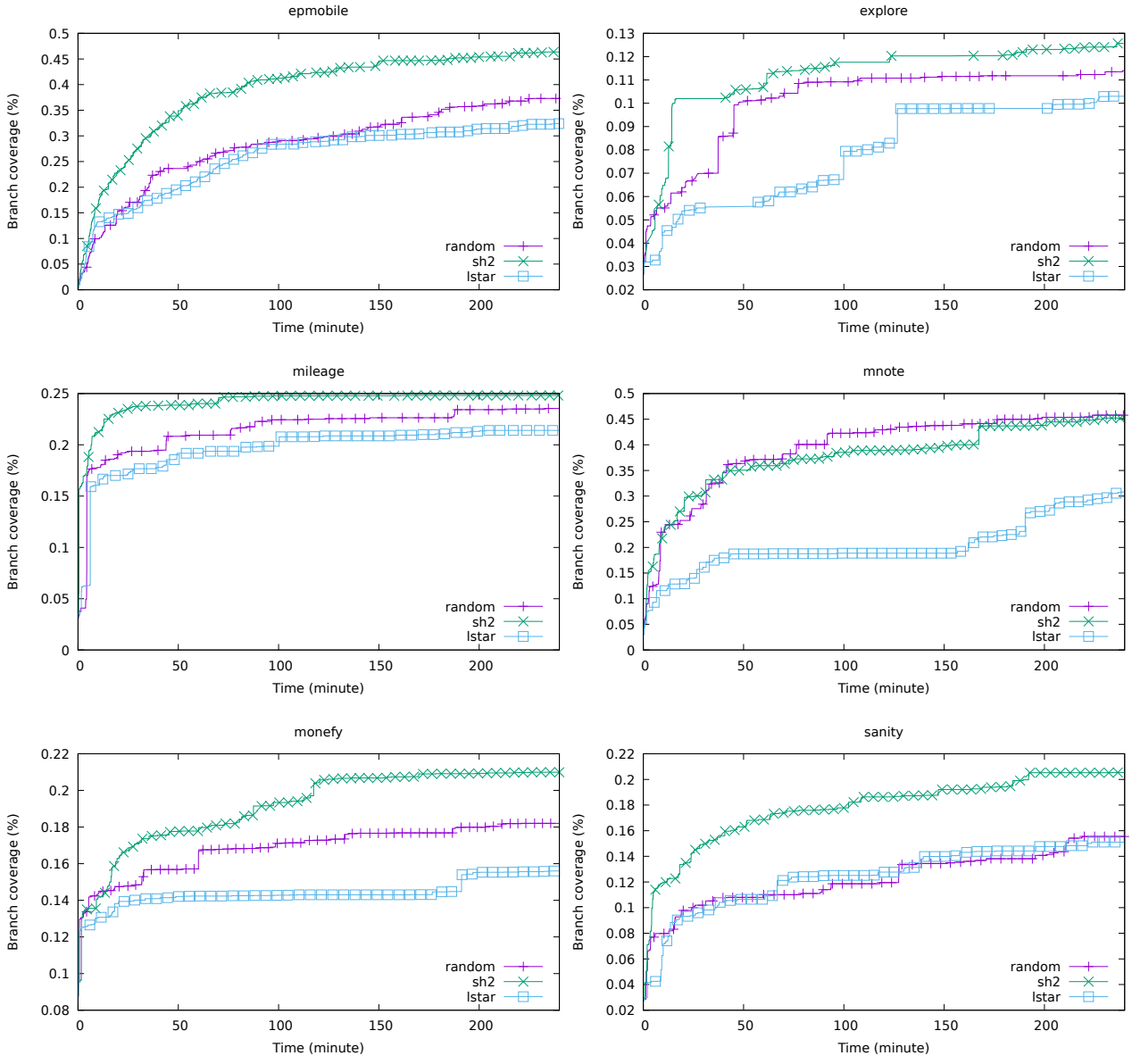


Figure 5.2: Comparison of the branch coverage of SwiftHand, Random and $\mathcal{L}^*$, #2

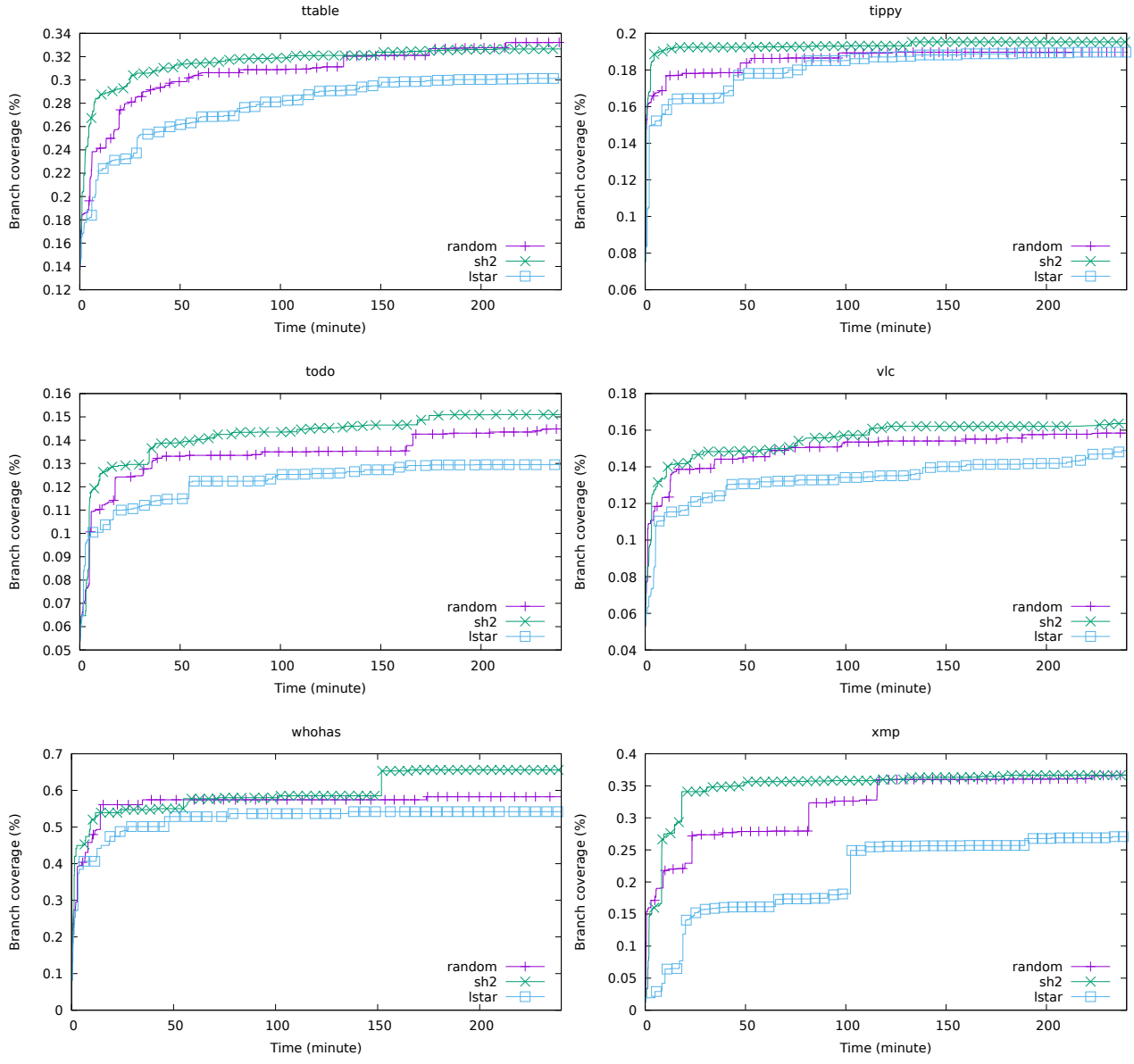


Figure 5.3: Comparison of the branch coverage of SwiftHand, Random, and $\mathcal{L}^*$, #3

App	% Br			% Time$_{reset}$			Resets / Inputs (%)			#Unique prefixes			# Scr		
	SH	Rand	$\mathcal{L}^*$	SH	Rand	$\mathcal{L}^*$	SH	Rand	$\mathcal{L}^*$	SH	Rand	$\mathcal{L}^*$	SH	Rand	$\mathcal{L}^*$
acar	21.46	17.89	14.60	21.59	34.65	50.50	1.80	18.09	16.63	5520	1788	2263	227	123	108
amoney	19.24	14.56	13.37	20.78	44.11	45.26	1.65	14.53	15.69	5743	3487	2151	194	99	58
amemo	45.66	36.83	31.01	22.34	37.52	48.39	1.87	11.53	16.89	5633	4568	2352	143	100	70
astrid	35.68	32.90	28.59	15.62	22.21	42.27	1.47	13.00	18.61	4395	2486	2071	223	129	78
cnote	36.58	36.64	30.09	20.23	29.62	51.07	1.76	8.91	17.96	6236	5818	2534	143	133	69
dmoney	46.81	43.36	39.61	22.91	14.62	49.73	1.60	8.65	17.21	5425	2510	2055	91	81	71
emobile	46.36	37.30	32.39	18.35	22.17	53.31	1.68	8.20	18.28	6500	5217	2749	261	197	170
explore	12.57	11.40	10.30	12.10	22.92	35.82	1.21	11.13	16.58	2694	2837	1728	98	69	43
mileage	24.81	23.55	21.41	18.15	19.43	51.45	1.64	9.02	20.53	5771	4068	2559	130	111	62
mnote	45.22	45.78	30.83	23.97	33.28	54.33	2.17	10.34	18.97	6212	5538	2822	127	106	51
monefy	20.99	18.20	15.60	21.74	35.79	60.08	1.68	8.95	19.43	6223	6388	2597	69	47	20
sanity	20.54	15.55	15.11	19.03	46.04	38.39	1.64	14.81	13.07	4968	3239	2027	179	93	98
ttable	32.66	33.22	30.14	20.07	35.17	50.78	1.74	11.68	17.95	5827	4614	2489	144	109	74
tippy	19.53	18.97	18.97	20.19	34.96	50.77	1.76	10.23	17.00	6415	6200	2634	21	19	17
todo	15.10	14.48	12.95	15.26	28.11	46.42	1.39	11.84	17.89	4760	3597	2169	81	56	36
vlc	16.36	15.84	14.89	18.13	33.44	48.33	1.60	11.66	17.42	5050	3626	2076	64	58	48
whohas	65.58	58.26	54.20	20.49	36.20	40.18	1.83	14.44	15.54	6272	3127	1935	28	21	23
xmp	36.68	36.66	27.07	20.71	27.46	45.34	1.78	8.33	14.45	6129	5358	2408	75	62	42

Table 5.2: Summary of comparison experiments: SwiftHand, Random, and $\mathcal{L}^*$

Comparison with Random and $\mathcal{L}^*$-based Testing

Table 5.2 summarizes the results of applying the three testing strategies to the eighteen Android apps. In the table, we use SH to denote SwiftHand. The *% Br* columns report branch coverage percentage for each strategy. *% Time$_{reset}$* shows the percentage of execution time spent on restarting the app. *Reset/Input* reports the ratio of the number of restarts to the number of inputs generated for each strategy. *#Unique prefixes* reports the number of unique input prefixes attempted in each testing strategy. *# Scr* reports the number of abstract screens covered by each strategy.

Figures 5.1, 5.2, and 5.3 plot the progress of the branch coverage in percentage against testing time.

- In all cases, SwiftHand achieves greater branch coverage than $\mathcal{L}^*$-based testing. SwiftHand also achieves more branch coverage than Random for twelve apps. For six apps (`cnote, mnote, ttable, tippy, vlc, xmp`), the Random strategy and SwiftHand performed equally well.

- For most of the apps, SwiftHand achieves branch coverage at a rate significantly faster than Random or $\mathcal{L}^*$-based testing. For example, in `amemo`, SwiftHand reaches almost 40% branch coverage within 40 minutes, whereas both Random and $\mathcal{L}^*$-based testing fail to reach 40% coverage in 240 minutes. This suggests that SwiftHand is superior for these apps when the time budget for testing is limited.

- Random restarts more frequently than SwiftHand. In comparison with desktop apps, Android apps have fewer GUI components on screen. Therefore, the probability is relatively high that Random will terminate an app by accidentally pushing the back

button or some other quit button. SwiftHand can circumvent this by merging states and by avoiding input sequences that lead to a terminal state.

- For some apps, Random catches up with SwiftHand given additional time budget. These apps can be categorized into two groups.
  - There are apps that cannot be tested efficiently by using only GUI-level inputs (such as touch and scroll). For example, if an app depends heavily on the contents of files, no testing strategy can reach deep states of the app without properly preparing the content of the file system. Among our benchmark apps, `explorer`, `mnote`, `vlc`, and `xmp` belong to this category. Similarly, if an app depends heavily on data values (i.e., numbers and texts), there is a limit to what can be achieved by considering only the GUI of the app; a more involved dynamic or static program analysis is necessary to handle such an app. In our benchmark set, `cnote`, `dmoney`, `ttable`, and `todo` belong to this category.
  - Several apps are just too simple (`mileage`, `tippy`, and `whohas`). For these apps, *any* testing strategy can achieve good test coverage.

  When an app is sufficiently complicated and can be tested efficiently by concentrating on the GUI (`acar`, `amemo`, `amoney`, `astrid`, `emobile`, `monefy`, and `sanity`), Random cannot achieve same coverage as SwiftHand.

- These experiments confirm the inadequacy of $\mathcal{L}^*$ as a testing strategy when restart is expensive, because $\mathcal{L}^*$ spends approximately half of its execution time restarting the app under test (Table 5.2). This results in low branch coverage over time (Figure 5.1 to 5.3). Furthermore, $\mathcal{L}^*$ has relatively low numbers in the *#Unique Input Prefixes* column compared with Random and SwiftHand. This indicates that $\mathcal{L}^*$ executes the same prefix more often than Random and SwiftHand.

What Restricts Test Coverage?

Manual analysis of apps with coverage lower than 30% reveals three key reasons for low branch coverage.

Combinatorial state explosion. `mileage` is a vehicle management app with a moderately complex GUI. The app provides a tab bar for easy context-switching between different tasks, and this tab bar creates a combinatorial state-explosion problem. Conceptually, each tab view is meant for a different independent task—most actions on a tab view affect the local state of the tab, with only few affecting the app's global state. SwiftHand does not understand this difference. As a result, even a small change in one tab view makes all other tabs treat the change as a new state in the model. The following diagram illustrates this situation:

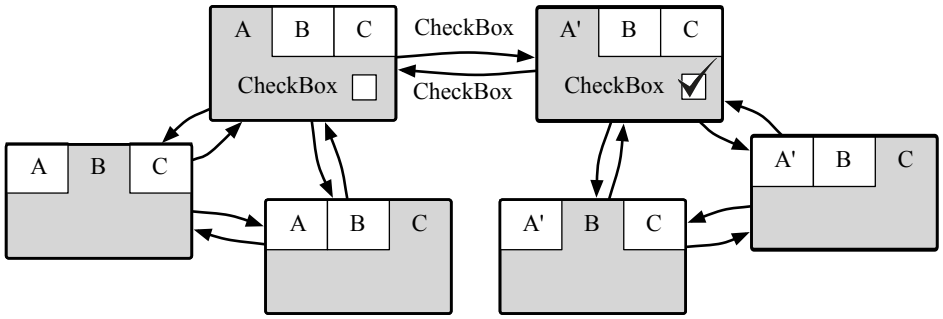


Figure 5.4: Model state explosion with tab view

The app `sanity` has a similar problem. This is an inherent limitation of our current model. We believe that this limitation can be overcome by considering a framework in which each model is a cross-product of several independent models. We could then perform compositional machine learning to learn the sub-models independently of one another. That said, at this point, it is not clear how this compositional learning can be done online.

Files and data values. As explained in Section 5.2, if apps depend on the content of files or data values then they cannot be tested effectively with a purely GUI-based testing strategy. For example, `xmp` is a media player app and cannot be tested thoroughly without video files to play. Similarly, `cnote` is a personal information manager app. Testing its search or editing features requires first filling the app with proper data.

Inter-app communication. `mnote` uses an intent to open a text file from file system navigation. When an intent is captured, the Android system pops up a system dialog for confirmation, but this is not part of the app itself. In such a situation, SwiftHand simply waits for the app to wake up and terminates it when there is no response. As a result, the viewer component of the app is never tested. The 45% code coverage results from testing only the file system navigation.

5.3 Evaluation of DetReduce

In this section, we evaluate the effectiveness of the DetReduce algorithm using test suites automatically generated by SwiftHand and Random. Specifically, we are interested in the size of the test suites reduced by DetReduce and the speed of the DetReduce algorithm.

We aim to answer the following research questions.

- **RQ1**: How effective is DetReduce in reducing test suites?
- **RQ2**: Does DetReduce run in a reasonable amount of time?

- **RQ3**: Does DetReduce preserve test coverage?
- **RQ4**: Does DetReduce preserve fault-detection capability?
- **RQ5**: How many re-executions are required to demonstrate the replayability of a trace?
- **RQ6**: What will happen to the splicing algorithm if the number of fragments in traces is increased beyond three?

To answer **RQ1** to **RQ5**, we generated test suites using two test generation algorithms (SwiftHand [23] and Random [23]) on eighteen benchmark apps and applied DetReduce to them. To answer **RQ6**, we analyzed the relationship between the likelihood of finding a replayable trace and the bound on the number of fragments in traces using four relatively complicated apps.

Experimental Setup

Generating a Deterministic Test Suite to be Used for Minimization

To generate a test suite to be used as inputs to DetReduce, we first collected execution traces by running an implementation of the SwiftHand [23] and Random [23] algorithms. We ran each for eight hours, then checked whether the generated traces are replayable by re-executing each trace ten times. For each non-replayable trace, we identified a non-empty replayable prefix of the trace and retained the prefix rather than throwing the entire trace away. Note that the traces used for evaluating DetReduce are different from the traces used for evaluating SwiftHand.

Avoiding non-replayable traces. The actual execution of an app can generate a non-replayable trace for several reasons: a) the app has external dependency (e.g., it receives messages from the outside world, depends on a timer, or reads and writes to the file system), b) the app uses animation that the underlying GUI testing library UIAutomator [3] cannot properly handle, or c) the app has inherent non-determinism due to the use of a random number generator or multi-threading. We eliminated dependency of an app on the outside world by resetting the contents of the SD card and the app data with every restart, and reduced the effect of animation on non-determinism by checking the contents of the GUI-tree at one-second intervals (which allowed us to check if a screen is still changing). However, it is nonetheless impossible to eliminate all sources of non-determinism. Therefore, we replayed each trace generated by the SwiftHand and Random algorithms ten times to remove the non-replayable suffixes of traces. We determined experimentally that eight re-executions is sufficient to detect most of non-replayable traces for the benchmark apps.

Why we did not use Monkey to generate the initial test suite?

Monkey [36] is a fuzz testing tool for Android apps. Monkey is widely-used to automatically find bugs. We initially attempted to use Monkey to generate inputs for DetReduce. However, we found that Monkey is not capable of generating replayable traces. We now describe our experience with Monkey.

Monkey is a simple black box tool that reports only the sequence of actions it used to drive a testing session. To get a trace would require non-trivial modifications to Monkey. Before jumping into this effort, we performed an experiment to determine whether Monkey is even capable of generating replayable traces or not—if Monkey cannot generate replayable traces, there is no point in the modification.

In this experiment, we used a script to generate traces with partial information from Monkey and checked if those traces could be replayed. The script injects user actions at the rate of m actions per second, collecting branch coverage and screen abstraction after injecting every n actions. The script picks the value of m from the set $\{1, 2, 5, 10, 100, 300\}$ and value of n from the set $\{2, 10, 50, 100, 200\}$. For each pair of values for m and n, the script runs Monkey until it injects 2000 actions. By combining the sequence of actions reported by Monkey with the collected coverage information, the script can generate traces that have coverage and screen information after every n actions (instead of having the information after every event.) We call such traces *partial traces*.

Using this script, we collected three partial traces for each possible value of m and n using the same random seed and checked if the partial traces are equal. If the partial traces do not match, this indicates that Monkey cannot generate a replayable trace. We performed the experiment using ten apps with three different random seeds.

The results of this experiment showed that Monkey passes the test for four apps when $n = 2$ and $m = 2$. For the other six apps, Monkey fails the test even when injecting one action per second. At this speed Monkey becomes useless in practice because its power primarily comes from its ability to inject many actions quickly. It will take too long to generate a sufficiently good test suite using Monkey at this speed. Therefore, we have concluded that using Monkey is not viable for generating the initial replayable test suite.

We found that Monkey injects actions asynchronously—that is, Monkey injects an action without checking whether the previously injected action has been fully handled or not. This allows Monkey to inject an order of magnitude more actions than testing tools that synchronously inject actions, but this also makes Monkey highly non-deterministic. For example, we noticed that if actions are injected while the app is unresponsive, those actions are dropped. Because the period of unresponsiveness varies from execution to execution, the number of dropped actions varies across executions. Note that this is an inherent problem with Monkey regardless of whether the target app is non-deterministic.

app	initial test suites				phase 1 results				phase 2 results				running time			
	#br	#sc	#act	#tr	#br	#sc	#act	#tr	#br	#sc	#act	#tr	t_{p1} (hr.)	t_{p2} (hr.)	t_r (hr.)	t_r/t
acar	4427	171	13478	822	4427	171	1808	170	4427	171	1283	121	11.86	8.31	0.90	11.22%
amemo	2955	114	13604	835	2955	114	1380	135	2955	135	1030	101	11	8.03	0.72	9.06%
amoney	4481	159	13213	779	4481	159	2793	269	4462	157	1595	146	11.27	12.07	1.14	14.21%
astrid	6201	170	10532	680	6201	170	1828	188	6201	170	1168	120	15.95	9.20	1.07	13.32%
cnote	5089	102	13584	157	5089	102	1515	156	5089	102	1083	117	12.3	7.84	1.09	13.61%
dmoney	2387	47	13511	785	2387	47	728	74	2387	47	574	63	5.9	3.53	0.43	5.45%
emobile	1554	214	13261	782	1554	214	1593	179	1554	214	1224	137	10.2	8.49	0.95	11.89%
explore	6647	105	7559	703	6647	105	1458	153	6596	103	867	92	19.35	10.57	1.07	13.42%
mileage	1766	81	13570	809	1766	81	507	61	1766	81	402	46	4.43	4.14	0.31	3.84%
mnote	889	76	13697	1003	889	76	988	96	889	76	718	71	9.14	7.59	0.47	5.92%
monefy	4966	62	13703	806	4966	62	2001	121	4966	62	1331	85	11.32	10.98	0.80	9.98%
sanity	978	186	12735	764	978	186	1639	142	978	186	1045	94	13.59	10.39	0.76	9.54%
tippy	712	15	14200	819	712	15	294	32	712	15	198	23	13.28	7.18	0.15	1.83%
todo	1415	58	10164	641	1415	58	735	82	1415	58	520	57	5.96	3.96	0.50	6.38%
ttable	2651	125	13028	1516	2651	125	1385	152	2251	125	891	97	9.71	10.25	0.53	6.36%
vlc	2341	60	11978	770	2341	60	719	76	2279	59	440	45	5.89	3.83	0.35	4.41%
whohas	230	15	12857	757	230	15	179	20	230	15	119	12	1.26	1.14	0.09	1.14%
xmp	2079	50	1326	761	2079	50	617	64	2079	50	342	34	3.98	2.27	0.28	3.53%
median	2333	91.5	13237	780.5	2333	91.5	1384	128	2333	91.5	879	88.5	9.96	7.92	0.63	7.84%

Table 5.3: Test suite reduction result using DetReduce on SwiftHand traces

DetReduce Configuration

In the evaluation, we configured the second phase of DetReduce to use at most three fragments to construct a trace. We also set both phases of DetReduce to re-execute each trace ten times to check the replayability of the trace. We intentionally set the number of re-executions conservatively to learn the relationship between the capability to detect non-replayable traces and the number of re-executions.

Evaluation of DetReduce

Table 5.3 and Table 5.4 show the results of applying DetReduce to the test suites generated by the SwiftHand and Random algorithms, respectively. Each table has four parts. The first part shows the following information about the test suites to be minimized: total branch coverage (#br.), total screen coverage (#s.), total number of transitions (#act.), and total number of traces (#tr.) of each initial test suite. The second and third parts of the table show information about the test suites generated after running the first and second phase, respectively, of DetReduce. The fourth part shows important statistics summarizing the experiment results: the running time of each phase of the algorithm (t_{p1} and t_{p2}), the execution time (t_r) of the resulting reduced regression test suite, and the ratio of the execution time of the resulting regression suite to the execution time of the original test suite in percentage (t_r/t). We make the following observations from the data shown in the tables:

- **RQ1**: The execution time of the reduced test suites (t_r) is several orders of magnitude shorter than that of the original test suites (eight hours). This shows that DetReduce

app	initial test suites				phase 1 results				phase 2 results				running time			
	#br	#sc	#act	#tr	#br	#sc	#act	#tr	#br	#sc	#act	#tr	t_{p1} (hr.)	t_{p2} (hr.)	t_r (hr.)	t_r/t
acar	2897	102	6990	2162	2897	102	943	96	2897	102	719	70	6.07	4.85	0.54	6.70%
amemo	2663	99	11680	1358	2663	99	1072	108	2663	99	768	74	10.06	5.4	0.48	6.03%
amoney	3285	110	10290	1406	3285	110	921	88	3261	106	671	66	6.81	4.61	0.42	5.32%
astrid	4797	112	7297	954	4797	112	1095	115	4797	112	760	79	17.01	6.38	0.66	8.22%
cnote	5000	88	12909	1140	5000	88	1252	123	5000	88	885	82	11.95	9.26	0.56	7.00%
dmoney	2057	46	7202	577	2057	46	567	59	2057	46	435	45	5.47	2.95	0.44	5.44%
emobile	1359	195	10500	847	1359	195	1276	153	1357	194	978	121	10.99	7.07	0.90	11.20%
explore	6145	76	5960	747	6145	76	913	86	6145	76	604	55	9.54	9.51	0.70	8.75%
mileage	1722	80	7013	670	1722	80	530	60	1722	80	344	44	4.74	3.19	0.39	4.84%
mnote	909	65	9559	1087	909	65	824	82	909	65	636	59	8.32	4.57	0.48	5.94%
monefy	3549	36	11435	970	3549	36	1449	78	3549	36	622	37	10.57	2.42	0.38	4.78%
sanity	701	110	8778	1350	701	110	706	66	701	110	433	42	4.64	3.84	0.31	3.93%
tippy	686	15	10999	1057	686	15	269	28	686	15	174	19	1.63	1.14	0.13	1.68%
todo	1312	39	7873	975	1312	39	557	76	1312	39	317	38	6.58	5.57	0.57	7.12%
ttable	2589	100	9242	1125	2589	100	1034	114	2589	100	730	71	17.56	14.48	0.32	3.96%
vlc	2001	44	7706	922	2001	44	528	62	2001	44	316	33	5.53	3.73	0.32	4.00%
whohas	206	44	7879	1179	206	16	141	19	206	16	81	11	1.2	0.75	0.08	0.98%
xmp	1798	45	9734	844	1798	45	566	48	1798	45	318	27	5.25	2,31	0.26	3.24%
median	2029	78	9269	1016	2029	78	868	80	2029	78	613	50	6.29	4.59	0.43	5.38%

Table 5.4: Test reduction result using DetReduce on Random traces

is highly effective in minimizing the test suites for the benchmark apps. Regarding the sizes of test suites, phase 1 of DetReduce removes 91.27% of transitions (median) and 90.5% of restarts (median). Phase 2 of DetReduce further removes 33.07% of transitions and 31.81% of restarts from the test suites obtained after Phase 1. These two phases of DetReduce cumulatively remove 93.84% of transitions and 93.52% of restarts. We also found that the rate of reduction is higher for test suites generated from Random. This is because these test suites have lower test coverage and more redundancies.

- **RQ2**: The running time of the algorithm is within a factor of 6 of the execution time of the original test suites generated by the test generation algorithms. More than half of the running time was spent in detecting and eliminating loops in phase 1 (and note that DetReduce spends no time removing redundant traces because those traces do not require any execution). The time spent in phase 1 is reasonable because the phase searches for a minimized test suite while eliminating redundant loops from each trace. Note that these experiments employed a conservative parameter (ten) for the number of re-executions to perform to check trace replayability, and the running cost of DetReduce can be further reduced by setting this parameter to eight. If we apply delta-debugging to minimize the test suites, the result will be a slowdown of several orders of magnitude. This is because delta-debugging executes a test suite $O(log(n))$ times in the best case, and $O(n^2)$ times in the worst case, where n is the number of transitions in the initial test suite.

- **RQ3**: Despite using an approximate method for checking if a trace a replayable, the minimized test suites nonetheless cover most of the original branch and screen coverage.

DetReduce fails to provide 100% coverage only for `amoney`. We manually analyzed the reasons for the missing branches and screens, and determined that non-replayable traces were not fully removed while generating the original test suites before phase 1 of DetReduce.

- **RQ4**: In order to check how DetReduce affects the fault-detection capability of test suites, we collected exceptions raised while executing each test suite. We identified seven distinct exceptions, and all survived after applying DetReduce. Note that DetReduce does not consider exceptions to be part of the test coverage it tries to preserve.

- **RQ5**: We measured how many re-executions were required to identify each non-replayable trace created during our experiments. The following table summarizes the results.

app	T	n=1	n=2	n=3	n=4	n=5	n=6	n=7	n=8	n=9	n=10
acar	558	90	5	7	4	0	4	1	0	0	0
amemo	893	459	4	6	1	1	1	0	0	0	0
amoney	787	280	3	3	5	3	5	3	0	0	0
astrid	1468	963	0	2	1	0	0	0	0	0	0
cnote	1433	942	5	5	1	1	2	0	0	0	0
dmoney	535	294	0	0	0	0	0	0	0	0	0
emobile	1630	518	2	2	1	5	0	0	0	0	0
explore	1440	450	14	7	2	4	1	2	1	0	0
mileage	500	280	3	0	3	3	0	0	0	0	0
mnote	1045	812	4	6	3	1	3	0	1	0	0
monefy	447	190	5	1	3	2	3	1	0	0	0
sanity	1665	1317	1	0	2	0	1	0	0	0	0
tippy	293	136	1	0	0	0	0	1	0	0	0
todo	1006	742	11	8	5	2	1	0	0	0	0
ttable	3009	2561	6	1	2	1	3	0	0	0	0
vlc	499	238	16	15	6	4	0	2	1	0	0
whohas	211	149	0	0	0	0	0	0	0	0	0
xmp	668	538	40	23	16	8	3	4	0	0	0
sum	18043	10956	120	86	55	34	27	14	3	0	0

The first column (app) shows the name of the app, the second column (T) shows the total number of traces created during the experiments, and the remaining columns show the number of non-replayable traces that required n re-executions for detection. The last row (sum) shows the summation of each column. The results show that all non-replayable traces were detected within the first eight iterations. The results also show that 39.3% of traces attempted during the experiments were replayable traces, suggesting that DetReduce is good at selecting candidate traces in our benchmarks.

app	#replayable traces (replay success rate)									
	k=1	k=2	k=3	k=4	k=5	k=6	k=7	k=8	k=9	k=10
acar	195	159	119	96	94	80	70	47	54	47
astrid	189	108	93	64	51	41	22	16	19	14
cnote	186	117	77	73	53	31	22	31	16	11
emobile	199	169	139	116	96	93	62	41	50	57

Table 5.5: The number of replayable traces out of 200 sampled traces.

Splicing and the Number of Fragments in Traces

RQ6: To understand the effect of bounding the number of trace fragments in phase 2 of our algorithm, we measured the relationship between the bound and the likelihood of finding a replayable trace, and the average number of trace fragments in a trace generated by splicing. For these measurements, we used four relatively complex benchmark apps.

Bounding the Number of Fragments and the Replayability of Traces

We measured the correlation between the bound on the number of fragments and the possibility of finding a replayable trace using ten different bounds on k ($1 \leq k \leq 10$). For each k, we constructed 200 random traces by combining k trace fragments from the test suite after phase 1. Furthermore, we restricted each trace to containing only 20 transitions. In order to construct the traces, we first collected at most 20,000 traces satisfying the requirement using breadth-first search of the transition system Q_{T_r} (described in section 3.2). Note that the paths of Q_{T_r} consist of traces that can be constructed via splicing. We then sampled 200 traces from the set of 20,000 traces. Finally, we checked how many of the sampled traces are replayable by executing each trace ten times. Table 5.5 shows the results. The first column shows the name of the app and the rest of the columns indicate the number of replayable traces for each k. Our hypothesis was that increasing the number of fragments would decrease the possibility of finding replayable traces, and the results confirm this hypothesis for the four apps.

Number of Fragments in Traces Generated by Splicing

A possibly simpler algorithm for generating a minimized test suite would be to construct a labeled transition system Q_T from T and then generate a minimal set of paths from Q_T so that these paths cumulatively attain full coverage. Such an algorithm would be equivalent to running phase 2 of DetReduce on the original test suite—that is, the algorithm would construct traces by combining trace fragments from T without imposing a bound on the number of trace fragments to be combined. Our hypothesis was that this algorithm would not scale because a significant number of traces generated using such an algorithm would contain many trace fragments, making them non-replayable with high probability. Therefore, the algorithm would spend considerable time checking the replayability of non-replayable traces. In order to validate this hypothesis, we constructed 1000 traces composed of at

app	#traces									
	k=1	k=2	k=3	k=4	k=5	k=6	k=7	k=8	k=9	k=10+
acar	25	14	22	28	37	24	42	48	73	689
astrid	29	3	15	16	10	13	23	38	70	783
cnote	12	4	11	2	7	8	10	6	13	927
emobile	25	16	20	23	24	59	33	59	88	685

Table 5.6: The number of traces composed of k fragments out of 1000 sampled traces

most 20 transitions by sampling random paths from Q_T, and checked the number of trace fragments in each sampled trace. Table 5.6 shows the results. The first column shows the name of the app and the rest of the columns indicates the number of traces composed of k fragments for each k between 1 to 10. The results show that there are many more traces composed of a larger number of fragments compared with traces composed of fewer fragments. Consequently, if we perform splicing without bounding the number of fragments, we are more likely to get traces composed of a large number of fragments. The results of the previous experiment (Section 5.3) suggest that such traces are likely to be non-replayable. This validates our hypothesis that a simple algorithm exploring the paths of Q_T without bounding the number of fragments in each trace will not scale.

Threats to Validity

We now discuss threats to the validity of our evaluations. We have identified three major issues.

Selection of apps We used a limited number of benchmark apps to evaluate DetReduce, so it is possible that our results do not generalize. To address this issue, we carefully selected the benchmark apps, and the details of the selection process are explained in Section 5.1.

Selection of test generation algorithms The selection of the test generation algorithms could potentially bias the evaluation results. Specifically, the results obtained from a single test generation algorithm cannot determine whether the results can be generalized to the other test generation algorithms. To address this, we used both SwiftHand and Random algorithm. We could not use Monkey because Monkey cannot generate replayable traces, as described in Section 5.3. The results obtained using Random show that DetReduce is not an artifact that works only with SwiftHand. However, the results are still not strong enough to decisively conclude that DetReduce can effectively reduce test suites generated from an arbitrary test generation tool.

The robustness of the fault detection capability evaluation In the evaluation, we checked whether the exceptions raised by the original test suites are also raised by the test suites reduced by DetReduce. However, this is a limited evaluation. A more robust

evaluation would involve injecting artificial faults into the benchmark apps and measuring the number of injected faults detected before and after the test suite reduction. We did not take this approach because of the difficulty of injecting faults into the binary of an Android app.

Chapter 6

Related Work

6.1 Automated GUI Testing

In this section, we compare automated GUI testing techniques with SwiftHand. Specifically, we consider four types of techniques: model-based testing, evolutionary-algorithms, fuzz testing, and symbolic execution.

Model-based Testing

Model-based testing approaches are often applied to testing graphical user interfaces [10, 17, 78, 96, 103, 113]. Such approaches model the behavior of the GUI abstractly using a suitable formalism such as event-flow graphs [78], finite state machines [86, 103], or Petri nets [96]. The models are then used to automatically generate a set of sequences of inputs (or actions), called a test-suite.

Memon et al. [8, 78, 119] proposed a two-staged automatic model-based GUI testing idea. In the first stage, a model of the target application is extracted by dynamically analyzing the target program. In the second stage, test cases are created from the inferred model. However, this approach differs from our work in two ways: First, in our research model creation and testing form a feedback loop. Second, we use *ELTS*, a richer model than *Event Flow Graph* (EFG). Yuan and Memon [120] also investigated a way to reflect runtime information on top of EFG model to guide testing. The two-staged model-based testing has also been combined with targeted exploration [13, 57], combinatorial testing [82, 87] and symbolic execution [81]. Azim and Neamtiu [13], Bhoraskar et al. [20], and Mirzaei et al. [82] proposed the use of static analysis to infer a behavior-model of an Android app to remove the need to construct models manually.

Crawljax [80] is an online model-leaning based testing technique for *JavaScript* programs. *Crawljax* tries to learn a finite state model of the program under test using state-merging. State-merging is based on the edit distance between DOMs. *Crawljax* can be seen as a learning-based GUI testing technique that uses ad-hoc criteria for state-merging. Recently, a number of model-learning based testing techniques targeting Android app [7, 93, 108, 117]

have been proposed. Yang et al. [117] take an online model-based testing approach similar to *Crawljax* [80]. They use static pre-analysis to refine the search space. Takala et al. [108] report a case study using online model-based testing. Rastogi et al. [93] proposed a testing technique based on learning a behavior-model of the target app. All of these techniques use some form of heuristic state-merging and are not based on formal automata learning. The main difference is in the treatment of counter-examples. Online automata learning techniques try to learn an exact model reflecting counter-examples. On the contrary, none of the above approaches learn from counter-examples. Therefore, these techniques may fail to learn the target model even for a finite state system, unlike online automata learning techniques. As far as we know, SwiftHand is the first GUI testing technique based on automata learning.

Ermuth and Pradel [30] recently proposed a technique to infer a set of meaningful action sequences from manually created test cases. The technique combines frequent subsequence mining [115] and inference of finite state machines [19] to infer meaningful action sequences. Once these sequences are inferred, one can construct longer action sequences using the inferred sequences as components. This technique can be a building block for other test generation techniques, including SwiftHand, although Ermuth and Pradel originally evaluated the inference technique using a random test generation method.

Evolutionary Algorithms

Evolutionary algorithms [14,127] have been used to test programs [102]. An approach based on an evolutionary algorithm generates a test suite that maximizes test coverage, and it can take a long time to generate such a test suite. Regarding GUI apps, EXSYST [42] first applied a genetic algorithm to GUI testing. EXSYST uses a set of test cases as an individual, and each test case as a gene. EvoDroid [70] is the first Android testing tool based on an evolutionary algorithm. Sapienz [73] uses a multi-object evolutionary algorithm to simultaneously maximize test coverage and minimize the size of the resulting test suite. These techniques outperform state-of-the-art fuzz testing tools such as *Monkey* [36] and Dynodroid [69] in terms of test coverage. However, it is not clear how much time is required to get such a good result.

Fuzz Testing

Monkey [36] is an automated fuzz testing tool. It creates random inputs without considering app's state. Hu and Neamtiu [52] developed a useful bug finding and tracing tool based on *Monkey*. Choudhary et al. [24] recently compared the performance of several automated testing tools for Android apps. Their results suggest that *Monkey* outperforms more sophisticated tools [8,13,23,47,69,117] in terms of maximizing coverage in a limited period of time. However, this observation does not conflict with our result from the SwiftHand experiment (Section 5.2), with the difference coming primarily from the speed of injecting actions. Our results indicate that a carefully designed model-learning based testing algorithm, such

as SwiftHand, can outperform a randomized algorithm if both inject actions at the same speed. In contrast, *Monkey* is capable of injecting two orders of magnitude more actions in the same period compared with the other testing tools. At the moment, sophisticated testing tools [8,13,23,47,69,117] cannot achieve the same speed because they need to collect information about the target app after injecting each action. However, the gap in injection speed can gradually be reduced by incorporating various optimization techniques, such as PCE [68].

MacHiry et al. [69] suggest a technique to infer a representative set of actions and perform action aware random testing. The idea is similar to the random strategy used in our experiment. Their action inference technique targets a larger action space including tilting and long-touching while our technique only considers touching and scrolling actions. Also, their tool acquires more runtime information by modifying Android framework to prune more actions at the expense of being less portable. On the contrary, SwiftHand modifies only the target app binary. Finally, they provide a comparison with *Monkey*. The result shows that *Monkey* needs significantly more time to tie the branch coverage of action aware random testing techniques.

Liang et al. [67] suggest a way to prioritize testing contexts when handling a multitude of apps. A testing context is the configuration of a testing environment, such as a network connection and a device to run a testing session. A key observation behind their approach is that if one testing context enabled a fuzz testing algorithm to find a bug in an app, then the same context tends to help finding a bug in similar apps as well. Thus, by first categorizing apps based on a similarity metric, one can plan a testing context prioritization strategy per category. This technique is orthogonal to our research, and SwiftHand can be used as a fuzz testing component of it.

Symbolic Execution

Anand et al. [9] applied concolic execution [33] to guide Android app testing. They use state subsumption checking to avoid repeatedly visiting equivalent program states. Their technique is relatively sound than our approach since concolic execution engine creates an exact input to find new branches in each iteration. In Mirzaei et al. [83]'s compiles Android apps to Java bytecode and applies *JPF* [50] with a mock Android framework to explore the state-space.

6.2 Model-learning and Testing

In this section, we compare model-learning algorithms and testing techniques based on model-learning with the SwiftHand algorithm.

Model-learning Algorithms

Angluin's $\mathcal{L}^*$ [12] is the most well-known active model-learning algorithm. The technique has been successfully applied to modular verification [26] and API specification inference [6]. However, the $\mathcal{L}^*$ algorithm and its descendants [54, 60, 98] are not adequate for guiding GUI testing for several reasons. First, these techniques try to learn a precise model for the target application, which is not necessary for testing. Second, they do not use a precise cost-model of learning queries, treating all queries as having the same execution cost. SwiftHand is specifically designed to be good at guiding GUI testing and so avoids these problems. Note that SwiftHand is not a variation of $\mathcal{L}^*$, although it is heavily inspired by $\mathcal{L}^*$; one should not use SwiftHand to replace $\mathcal{L}^*$, because SwiftHand is not designed to efficiently learn an exact model. We refer readers to the introductory article by Vaandrager [109] for more details regarding active model-learning algorithms and their applications.

Passive learning techniques [29, 65] do not assume the presence of a teacher and uses positive and negative examples to learn a model. François et al. [27] have introduced ideas of exploiting domain knowledge to guide passive learning. The technique was subsequently improved by Lambeau et al. [63].

Learning-based Testing

Machine learning has previously been used to make software testing effective and efficient [18, 40, 41, 43, 75, 88, 91, 111, 112, 114]. Meinke and Walkinshaw's survey [76] provides a convenient introduction to the topic. In general, classic learning-based testing techniques aim to check whether a program module satisfies a predefined functional specification. Also, they use a specification and a model checker to resolve equivalence queries. On the contrary, SwiftHand tries to maximize test coverage and actually executes a target program to resolve equivalence queries.

Meinke and Sindhu [75] reported that learning algorithms similar to $\mathcal{L}^*$ are inadequate as a testing guide. They proposed an efficient active learning algorithm for testing reactive systems, which uses a small number of membership queries.

6.3 Test Reduction Techniques

In this section, we discuss test reduction techniques compared with DetReduce. Specifically, we consider GUI test case minimization algorithms, delta-debugging, and test suite reduction algorithms.

GUI Test Minimization Techniques

ND3MIN [25] is the most closely related work to our research. It is a GUI test minimization algorithm for Android apps, based on delta-debugging [124]. We compare ND3MIN and

DetReduce in three aspects: their goals, the way they handle non-determinism, and their running time.

- Goals: ND3MIN aims to minimize each test case in isolation while keeping the final activity[1] that the test case reaches at the end. DetReduce tries to minimize a whole test suite while keeping the branch and screen coverage of the test suite. The tools have different goals.
- Handling non-determinism: ND3MIN aims to generate a test case that reaches the desired activity with a high probability, even if non-deterministic behaviors occur during the execution. On the contrary, DetReduce is designed to actively detect and avoid non-deterministic behaviors during the process of minimization.
- Running time: ND3MIN is a variation of delta-debugging. The worst case time complexity of delta-debugging is $O(n^2)$ where n is the size of input test case [124]. This time complexity could make the algorithm fail to scale because the cost of performing each test run is expensive in GUI testing. ND3MIN uses up to 50 hours to minimize a test case composed of 500 actions. DetReduce is capable of handling a test suite composed of more than 10,000 transitions in less than 30 hours.

Hammoudi et al. [45] also proposed a delta-debugging based test minimization algorithm. Unlike ND3MIN and DetReduce, their work aims to minimize manually written test cases for web apps. Their results showed that the execution time of the minimized test cases is on average 22% shorter than that of the original test cases. This shows that there is room to minimize even manually written test cases. Since they used relatively small test cases composed of less than 150 actions, it is hard to say if their delta-debugging approach would scale on a large GUI test case.

Test Suite Reduction Techniques

Test suite reduction techniques [46, 51, 55, 56, 59, 74, 104, 105, 118, 126] automatically reduce the size of a test suite without losing the coverage of the test suite. Unlike our work, these techniques assume that test suites consist of already compacted test cases; these techniques do not focus on reducing the size of each test case. They only focus on selecting a small set of test cases from a test suite. The first part of the first phase of DetReduce, where we remove redundant traces, can be seen as a test suite reduction technique. In the context of GUI testing, McMaster and Memon [74] proposed call-stack history as a metric for reducing GUI test suites. We might be able to reduce redundant traces more efficiently by adopting this technique. However, it is possible that removing too many traces at the first phase of DetReduce might negatively affect the capability of the second phase of DetReduce.

[1] Android apps are composed of a number of activities. Each activity represents a different functionality.

Delta-debugging

Delta-debugging [122, 124] is probably the most widely-known test minimization technique. It was proposed in the context of fault isolation and has been successfully applied to many problems [22, 84, 99, 123]. We found it difficult to use delta-debugging to minimize a large GUI test suite because of the cost of running the test suite.

It might be possible to make delta-debugging scale better on GUI test suites by incorporating domain specific knowledge. Delta-debugging performs a partitioning based search, and it is often possible to accelerate the partitioning based search by incorporating domain specific knowledge. For example, hierarchical delta-debugging (HDD) [84] works on structured texts, such as XML, by first performing delta-debugging on top-level structures, then gradually moving into substructures. This allows HDD to significantly reduce the time required to reduce structured texts compared to the original delta-debugging. A similar idea has been used in DEMi [100] to minimize test cases for a distributed system. However, we have yet to find a way to make delta-debugging scale better on GUI test suites.

6.4 Test Generation Techniques

In this section, we discuss test generation techniques focused on how they are related to DetReduce.

Automated GUI Testing Techniques

In this thesis, we used SwiftHand [23] to generate test suites as inputs for DetReduce. One can use any automated GUI testing technique, such as A3E [13], Dynodroid [69], AppsPlayground [93], or MobiGUITAR [8], to generate initial test suites. Automated testing techniques can also be directly used to find crashing bugs [94, 121], responsiveness bugs [89, 116], concurrency bugs [21, 71], and security problems [5, 15, 39, 92]

One may argue that test minimization might not be necessary for the future if automated testing techniques continue to improve. Automated GUI testing techniques are indeed becoming better in maximizing test coverage and finding bugs in a limited period of time [30,73]. However, the results of Hammoudi et al. [45] show that even test cases generated by human experts are reducible. The study by Yoo et al. [118] also suggests that even a test suite maintained by human testers needs to be compacted. Therefore, we predict that GUI test suite minimization techniques will remain useful, even though automated GUI testing techniques continue to improve.

Monkey

A recent survey [24] compares the performance of several automated testing tools for Android apps. Their results suggest that *Monkey* outperforms other more sophisticated tools in terms of maximizing coverage in a limited period of time. However, we observed that it is difficult

to replay test cases generated by *Monkey*. Even if *Monkey* finds a bug, it might be difficult to reproduce the bug or minimize the sequence of actions obtained from *Monkey*. [107]

Test scripts and Record-and-replay

GUI test scripts [37, 38, 95] and record-and-replay frameworks [34, 38, 44, 53, 90] are tools to generate reusable test cases reflecting human knowledge. These tools complement our approach. One can use these tools either to generate a set of test cases to be minimized or to add more test cases to already minimized regression test suites.

Chapter 7

Conclusion

This chapter summarizes the main technical contributions of the research in this thesis and addresses limitations and potential future work.

7.1 Summary

In this thesis, we presented SwiftHand, an automated GUI test generation algorithm for Android apps for which we do not have an existing model of the GUI. SwiftHand achieves code coverage quickly by learning and exploring a model of the app's GUI. Specifically, SwiftHand uses an extended labeled transition system that models how the app's GUI responds to user inputs. Using model-learning for test generation is not new [76], and existing algorithms typically employ the $\mathcal{L}^*$ [11] learning algorithm or one of its variants [98, 101]. However, $\mathcal{L}^*$-based test generation algorithms are inefficient in testing Android apps because $\mathcal{L}^*$ frequently restarts the app under testing in order to learn a precise model of the app. This results in meaningfully diminished performance because restarting is an expensive operation. SwiftHand instead performs an approximate learning in order to avoid frequent restart. We evaluated SwiftHand using eighteen Android apps, for which SwiftHand achieves better branch coverage than $\mathcal{L}^*$ or a randomized test generation algorithm. Even when an app is simple or an app cannot be tested efficiently using a purely GUI-based testing approach, when the branch coverage gap between the randomized algorithm and SwiftHand becomes less significant, SwiftHand achieves maximum branch coverage more quickly than the randomized algorithm. This evaluation also confirmed that the $\mathcal{L}^*$-based test generation algorithm is inadequate for testing Android apps because of frequent restart.

We have also presented DetReduce, an algorithm that automatically constructs a small regression test suite by reducing a large test suite generated by an automated test generation tool. An automated test generation tool tends to generate a test suite that is difficult to reuse and understand because of its large size. However, through manual analysis we have identified three kinds of common redundancy in automatically generated test suites: redundant test cases, redundant loops, and redundant subsequences. We also observed that some of these

redundancies could be removed without decreasing the coverage of the test suites. Based on this observation, we created DetReduce, a greedy algorithm to reduce automatically generated GUI test suites by removing these three kinds of redundancies. Being specialized to handle GUI test suites, DetReduce can reduce a large test suite more quickly than general-purpose test reduction algorithms, such as delta-debugging. Evaluation of DetReduce using eighteen Android apps revealed that DetReduce is capable of reducing the running time of the test suites generated with SwiftHand by an order of magnitude. We also observed that the test suites reduced by DetReduce retain all distinct exceptions raised while executing the original test suites.

7.2 Discussion

Although the techniques proposed in this thesis can help with testing GUI apps by automatically generating test cases, there are limitations and open problems whose resolution would make these techniques more broadly usable. Here, we address such limitations and open problems.

Limitation: Apps that Cannot Be Handled Efficiently

Both SwiftHand and DetReduce make several assumptions about the app to be tested: a) that the app can be restarted to bring it back to its initial state, b) that GUI-level user inputs (such as touch and scroll actions) are sufficient to test the app, c) that when the app is in a stable state, it is possible to infer a set of enabled user inputs by analyzing the GUI component tree of the app, and d) that the app is deterministic. In the real world, not every app satisfies these assumptions. In this section, we discuss several categories of apps that our algorithms cannot handle efficiently because one or more of the above assumptions is violated.

Non-determinism in Android apps. Many Android apps exhibit non-deterministic behaviors, for one of several reasons. The four primary sources of non-determinism are built-in randomness, time dependency, concurrency, and external dependency.

- Randomness appears mostly in games, but non-game apps also use randomness—for example, to control the frequency of exposing advertisements or to provide a randomized quiz. To test such an app, every random number generator in the app must be controlled to produce a deterministic sequence of numbers. This can be achieved by replacing every usage of random number generators using a binary instrumentation tool. We do not automatically modify apps in this manner because of the implementation overhead.

- Some apps depend heavily on the current time and date. For example, a calendar app shows different content depending on the date. To test such an app, every access to

the time and date must be controlled to consistently provide the same artificial value. Somewhat surprisingly, and unlike controlling random number generators, controlling the time and date value is a non-trivial exercise. One might think that it is sufficient to reset the system clock to a pre-configured value during app tests, but this will not work if the target app has an additional source (such as a remote server) from which to access the time and date. The situation becomes even more complicated if timers are involved. For example, an app might use a timer to show an alert ten seconds after the app is executed. If we cannot fully control the speed of test execution, the alert might appear on a different screen every time we execute a test case— on one run, we might be able to inject ten events in ten seconds, and on another run eleven. In some sense, the existence of a timer poses a real-time constraint on executing a test case. Overall, dealing with time dependency remains an open problem.

- In a GUI app, concurrency is extensively used to improve responsiveness. It is recommended to delegate long-running computations to worker threads, and not to perform anything demanding at the main GUI thread of the app. To faithfully replay an execution of a concurrent program, we must record the logical ordering among execution steps of multiple threads and replay the exact order. This means that a proper test case for a concurrent program must include the thread ordering information for it to be deterministically replayed. In the case of an Android app, obtaining thread-ordering information and enforcing it requires instrumenting the app binary and modifying the Android framework. However, the current implementations of SwiftHand and DetReduce do not provide such mechanisms because of the implementation overhead.

- Smartphone apps often feature an external dependency, such as communication with remote servers. If data sent from a sever change over time, it can lead to non-deterministic behavior in an app communicating with that server. It is therefore preferable to use mock servers sending consistent data to test such apps, instead of connecting to real servers. Similarly, some apps use servers as storage. To restart such an app, an app-specific script to erase remotely stored data must be executed before running every single test case. Without providing mock servers and an app-specific restart script, an app depending heavily on remote severs will appear to be highly non-deterministic to SwiftHand and DetReduce.

Although both SwiftHand and DetReduce are designed to work with mostly deterministic apps, we implemented fallback mechanisms to handle when non-determinism occurs. When a non-determinism is detected, the implementation of SwiftHand reflects it to the learned model as a non-deterministic transition, and uses this information to avoid triggering non-deterministic transitions learned so far. DetReduce detects non-deterministic traces by re-executing all traces multiple times, and tries to keep deterministic prefixes of the detected non-deterministic traces. Even with these treatments, we found that neither algorithm (particularly DetReduce) works well if the target app is highly non-deterministic.

Developing testing algorithms that can effectively handle non-deterministic apps remains an open problem.

Files and strings. Smartphone apps often access files, and the execution of these apps depends heavily on the contents of the files. To test such an app, one must therefore prepare files with the proper content. Similarly, an app might take a string from a text input field and determine what to do next based on the contents of the string. Any testing algorithm based purely on GUI-level user inputs (such as touch and scroll actions) will face difficulty in testing such apps. To the best of our knowledge, automatically testing a GUI app handling files and strings remains an open problem. To resolve this, we would need to combine existing GUI test generation techniques with symbolic execution [61] or search-based testing techniques [48, 49], in the sense that hybrid concolic testing [72] combines random testing and concolic execution [33]. We could start by first using a GUI-testing technique to explore the execution space and identify a sequence of user inputs that leads the execution to a point where files and strings are used, and it is sufficient to employ a limited set of pre-defined strings and files at this stage. Once a specific input sequence is found, we could switch to symbolic execution or a search-based method to explore various files and strings. Note this is a limitation of SwiftHand and not a problem for DetReduce.

Inter-app communication. An Android app can communicate with another Android app running on the same device using intent communication[1]. An intent is used when an app wants to delegate a task to a different app. For example, most apps use intent communication to take a photo or to send an email. An app can start intent communication by first creating an intent and sending it to the Android framework. The Android framework, then, identifies and starts the appropriate app to delegate the intent. While the intent is being handled, the app that initiated the intent goes to the background, and the app receiving the intent comes to the foreground and takes the screen. Once the app that received the intent finishes the requested task, it returns execution control and the intent handling result to the original app. However, the current implementations of SwiftHand and DetReduce do not follow intent communications. The algorithms themselves do not have difficulty handling intent communication—rather, the difficulty lies in implementing a logic to detect whether a target app is terminated. The current implementation checks the package name of the active app (that is, the app holding the screen) to determine whether the target app has been terminated, and this method cannot distinguish temporarily inactive apps from terminated apps. To precisely track whether an app is still live requires access to the full activity stack and the intent communication history. Acquiring this information necessitates modifying the Android framework. We did not adopt this approach because of the implementation overhead and because this limitation does not restrict the choice of apps we can test. However, it does limit the maximum test coverage that SwiftHand can achieve.

[1] `https://developer.android.com/reference/android/content/Intent.html`

WebView. The current implementations of SwiftHand and DetReduce ignore the contents of WebView[2] components. This is because UIAutomator, the underlying low-level GUI testing library used in our implementation, offers only limited support for WebView. In Android, a WebView component is a wrapper of the WebKit[3] rendering engine, which is used to embed the contents of a web page into an app. The contents of a WebView component are stored internally as a DOM tree[4]. Unfortunately, UIAutomator provides only partial information about DOM trees. This prevents us from correctly inferring a set of enabled inputs. For example, UIAutomator does not provide the `z-index` and `display` attributes of DOM elements. Both attributes are frequently employed to control the visibility of DOM elements, and not knowing the values for these attributes prevents us from inferring a set of enabled inputs correctly. As such, our current implementation ignores the contents of WebView components. Although we do not have a solution for this problem, it limits only the maximum test coverage that SwiftHand can achieve and it does not restrict the breadth of apps for testing.

Games. SwiftHand and DetReduce cannot test games, and we do not plan to support games in the future. The most prominent reason for this is that games do not use a GUI library developed to support typical apps. Instead, games have their own GUI system internally and use only a single rectangular widget from an existing GUI library as a canvas for rendering a snapshot of the internal GUI. As such, to a GUI testing tool—which does not understand how games internally manage GUIs—games appear to have only a single canvas widget. Therefore, testing a game at the GUI level requires building a new testing infrastructure that understands how a game builds and manipulates its GUI. Further, games also involve additional difficulties discussed above: concurrency, time dependency, external dependency, and files.

Future Work

In the previous section, we discussed major limitations of SwiftHand and DetReduce. In this section, we discuss broad open problems in automated GUI test generation and propose future work.

Incorporating human knowledge. Although automated test generation tools are good at quickly generating a large number of test cases, they cannot replace human testers. Rather, human testers and automated tools should complement each other. Human testers are far more proficient than automated test generation tools for certain types of tasks—for example, a human tester can easily recognize a login screen during testing and quickly pass through by providing a proper credential. An automated test generation tool, on the other hand, will

[2] `https://developer.android.com/reference/android/webkit/WebView.html`
[3] https://webkit.org/
[4] `https://www.w3.org/DOM`

not be able to pass the login screen without being pre-programmed to behave properly when facing this screen. The pre-programming approach does not scale because login screens of different apps will look different and function differently. A more effective approach would be to provide an interface through which a human tester can transfer knowledge about such special cases to an automated test generation tool. A simple method would be for the testing tool to receive a manually crafted set of test cases from a human tester. This approach would allow SwiftHand to reach deep program states. However, DetReduce would gain no advantage from this approach. A more advanced approach would be to infer a set of meaningful subsequences from a set of manually created test cases. This would benefit both SwiftHand and DetReduce. SwiftHand could use the inferred sequences of actions as building blocks for constructing test cases, while DetReduce could maintain the occurrence of these sequences as much as possible while reducing test cases to avoid non-replayable test cases.

Test oracle. In order to achieve fully automated testing of a GUI app, an automated test generation technique alone is not sufficient because a human tester must nonetheless manually determine whether each test case diverges from the expectation. Automated test oracle [97] is a technique to mitigate this problem. The role of a test oracle is to automatically check the execution of test cases and to report when the execution diverges from the expectation. One way to implement an automated test oracle is to export test cases to executable test scripts. Once a test case is materialized as a script, a human tester can implement a test oracle by adding assertions to the script. Typically, GUI test oracles extract values or information from GUI components rendered on the screen [77], but this can result in scalability issues because all test scripts must be modified and maintained manually. Another option is to use a monitoring-oriented programming (MOP) framework [79], such as JavaMOP [58] or RV-Droid [31]. MOP frameworks allow a programmer to inject runtime specification checks into an existing program. Once specification checks are injected, the app will automatically check whether the specification is violated at runtime. This approach saves a human tester from the manual labor of modifying test scripts, but also has a higher entry barrier of implementing specification-checking using a MOP framework. Another interesting open problem is to develop a GUI app specific test oracle. For example, many GUI apps have a fragile GUI layout problem; the layout is fragile if the layout does not render correctly on certain screen resolutions. Checking the fragility of a GUI layout is a labor-intensive task because it must be tested manually on various resolutions.

Bibliography

[1] AppBrain Android Statistics: Number of Android applications. https://www.appbrain.com/stats/number-of-android-apps.

[2] StackOverflow Developer Survey Result 2017. https://insights.stackoverflow.com/survey/2017.

[3] UI Automator. https://google.github.io/android-testing-support-library/docs/uiautomator/.

[4] axml, read and write Android binary xml files. http://code.google.com/p/axml/, 2012.

[5] A. Abraham, R. Andriatsimandefitra, A. Brunelat, J.-F. Lalande, and V. V. T. Tong. GroddDroid: a gorilla for triggering malicious behaviors. In *Malicious and Unwanted Software (MALWARE), 2015 10th International Conference on*, pages 119–127. IEEE, 2015.

[6] R. Alur, P. Cerný, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. In *POPL*, pages 98–109, 2005.

[7] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. D. Carmine, and A. M. Memon. Using GUI ripping for automated testing of Android applications. In *ASE*, pages 258–261, 2012.

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

[9] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *SIGSOFT FSE*, page 59, 2012.

[10] A. A. Andrews, J. Offutt, and R. T. Alexander. Testing web applications by modeling with FSMs. *Software and System Modeling*, 4(3):326–345, 2005.

[11] D. Angluin. Inference of reversible languages. *J. ACM*, 29(3):741–765, July 1982.

[12] D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[13] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of Android apps. In *ACM SIGPLAN Notices*, volume 48, pages 641–660. ACM, 2013.

[14] T. Back. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford university press, 1996.

[15] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky. AppGuard–fine-grained policy enforcement for untrusted Android applications. In *Data Privacy Management and Autonomous Spontaneous Security*, pages 213–231. Springer, 2014.

[16] E. Balas. The prize collecting traveling salesman problem. *Networks*, 19(6):621–636, 1989.

[17] F. Belli. Finite-state testing and analysis of graphical user interfaces. In *12th International Symposium on Software Reliability Engineering (ISSRE'01)*, page 34. IEEE Computer Society, 2001.

[18] T. Berg, O. Grinchtein, B. Jonsson, M. Leucker, H. Raffelt, and B. Steffen. On the correspondence between conformance testing and regular inference. In *FASE*, pages 175–189, 2005.

[19] I. Beschastnikh, Y. Brun, J. Abrahamson, M. D. Ernst, and A. Krishnamurthy. Unifying FSM-inference algorithms through declarative specification. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 252–261. IEEE, 2013.

[20] R. Bhoraskar, S. Han, J. Jeon, T. Azim, S. Chen, J. Jung, S. Nath, R. Wang, and D. Wetherall. Brahmastra: Driving apps to test the security of third-party components. In *USENIX Security*, pages 1021–1036, 2014.

[21] P. Bielik, V. Raychev, and M. Vechev. Scalable race detection for Android applications. In *ACM SIGPLAN Notices*, volume 50, pages 332–348. ACM, 2015.

[22] J.-D. Choi and A. Zeller. Isolating failure-inducing thread schedules. In *ACM SIGSOFT Software Engineering Notes*, volume 27, pages 210–220. ACM, 2002.

[23] W. Choi, G. Necula, and K. Sen. Guided GUI testing of Android apps with minimal restart and approximate learning. In *ACM SIGPLAN Notices*, volume 48, pages 623–640. ACM, 2013.

[24] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for Android: Are we there yet?(e). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 429–440. IEEE, 2015.

[25] L. Clapp, O. Bastani, S. Anand, and A. Aiken. Minimizing GUI event traces. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 422–434. ACM, 2016.

[26] J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. Learning assumptions for compositional verification. In *TACAS*, pages 331–346, 2003.

[27] F. Coste, D. Fredouille, C. Kermorvant, and C. de la Higuera. Introducing domain and typing bias in automata inference. In *ICGI*, pages 115–126, 2004.

[28] C. De La Higuera, J. Oncina, and E. Vidal. Identification of DFA: Data-dependent versus data-independent algorithms. *Grammatical Interference: Learning Syntax from Sentences*, pages 313–325, 1996.

[29] J. N. Departarnento and P. Garcia. Identifying regular languages in polynomial. In *Advances in Structural and Syntactic Pattern Recognition, volume 5 of Series in Machine Perception and Artificial Intelligence*, pages 99–108. World Scientific, 1992.

[30] M. Ermuth and M. Pradel. Monkey see, monkey do: effective generation of GUI tests with inferred macro events. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 82–93. ACM, 2016.

[31] Y. Falcone, S. Currea, and M. Jaber. Runtime verification and enforcement for Android applications with RV-Droid. In *International Conference on Runtime Verification*, pages 88–95. Springer, 2012.

[32] M. Fazzini, E. N. d. A. Freitas, S. R. Choudhary, and A. Orso. From manual Android tests to automated and platform independent test scripts. *arXiv preprint arXiv:1608.03624*, 2016.

[33] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, pages 213–223, 2005.

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

[35] Google Inc. Google Play. `https://play.google.com/store?hl=en`, 2008. Accessed: 2017-04-11.

[36] Google Inc. UI/Application Exerciser Monkey. `https://developer.android.com/studio/test/monkey.html`, 2008. Accessed: 2017-04-11.

[37] Google Inc. MonkeyRunner. `https://developer.android.com/studio/test/monkeyrunner/index.html`, 2010. Accessed: 2017-04-11.

[38] Google Inc. Espresso. `https://google.github.io/android-testing-support-library/docs/espresso/`, 2011. Accessed: 2017-04-11.

[39] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1025–1035. ACM, 2014.

[40] A. Groce, D. Peled, and M. Yannakakis. AMC: An adaptive model checker. In *CAV*, pages 521–525, 2002.

[41] A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. *Logic Journal of the IGPL*, 14(5):729–744, 2006.

[42] F. Gross, G. Fraser, and A. Zeller. Exsyst: search-based GUI testing. In *Proceedings of the 34th International Conference on Software Engineering*, pages 1423–1426. IEEE Press, 2012.

[43] R. Groz, M.-N. Irfan, and C. Oriat. Algorithmic improvements on regular inference of software models and perspectives for security testing. In *ISoLA (1)*, pages 444–457, 2012.

[44] M. Halpern, Y. Zhu, R. Peri, and V. J. Reddi. Mosaic: cross-platform user-interaction record and replay for the fragmented Android ecosystem. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, pages 215–224. IEEE, 2015.

[45] M. Hammoudi, B. Burg, G. Bae, and G. Rothermel. On the use of delta debugging to reduce recordings and facilitate debugging of web applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 333–344. ACM, 2015.

[46] D. Hao, L. Zhang, X. Wu, H. Mei, and G. Rothermel. On-demand test suite reduction. In *Proceedings of the 34th International Conference on Software Engineering*, pages 738–748. IEEE Press, 2012.

[47] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan. Puma: Programmable UI-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pages 204–217. ACM, 2014.

[48] M. Harman. The current state and future of search based software engineering. In *2007 Future of Software Engineering*, pages 342–357. IEEE Computer Society, 2007.

[49] M. Harman and B. F. Jones. Search-based software engineering. *Information and software Technology*, 43(14):833–839, 2001.

[50] K. Havelund and T. Pressburger. Model checking Java programs using Java pathfinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):366–381, 2000.

[51] H.-Y. Hsu and A. Orso. Mints: A general framework and tool for supporting test-suite minimization. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 419–429. IEEE, 2009.

[52] C. Hu and I. Neamtiu. A GUI bug finding framework for Android applications. In *SAC*, pages 1490–1491, 2011.

[53] Y. Hu, T. Azim, and I. Neamtiu. Versatile yet lightweight record-and-replay for Android. In *ACM SIGPLAN Notices*. ACM, 2015.

[54] M. Isberner, F. Howar, and B. Steffen. The ttt algorithm: A redundancy-free approach to active automata learning. In *RV*, pages 307–322, 2014.

[55] D. Jeffrey and N. Gupta. Test suite reduction with selective redundancy. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 549–558. IEEE, 2005.

[56] D. Jeffrey and N. Gupta. Improving fault detection capability by selectively retaining test cases during test suite reduction. *IEEE Transactions on software Engineering*, 33(2), 2007.

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

[58] D. Jin, P. O. Meredith, C. Lee, and G. Roşu. JavaMOP: Efficient parametric runtime monitoring framework. In *Proceedings of the 34th International Conference on Software Engineering*, pages 1427–1430. IEEE Press, 2012.

[59] J. A. Jones and M. J. Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on software Engineering*, 29(3):195–209, 2003.

[60] M. J. Kearns and U. V. Vazirani. *An introduction to computational learning theory*. MIT press, 1994.

[61] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

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

[63] B. Lambeau, C. Damas, and P. Dupont. State-merging DFA induction algorithms with mandatory merge constraints. In *ICGI*, pages 139–153, 2008.

[64] K. Lang, B. Pearlmutter, and R. Price. Results of the Abbadingo One DFA learning competition and a new evidence-driven state merging algorithm, 1998.

[65] K. J. Lang. Random DFA's can be approximately learned from sparse uniform examples. In *COLT*, pages 45–52, 1992.

[66] D. Lee and M. Yannakakis. Principles and methods of testing FSMs: A survey. *Proc. of IEEE*, 84:1089–1123, 1996.

[67] C.-J. M. Liang, N. D. Lane, N. Brouwers, L. Zhang, B. F. Karlsson, H. Liu, Y. Liu, J. Tang, X. Shan, R. Chandra, et al. Caiipa: Automated large-scale mobile app testing through contextual fuzzing. In *Proceedings of the 20th annual international conference on Mobile computing and networking*, pages 519–530. ACM, 2014.

[68] F. X. Lin, L. Ravindranath, S. Nath, and J. Liu. SPADE: Scalable app digging with binary instrumentation and automated execution.

[69] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for Android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 224–234. ACM, 2013.

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

[71] P. Maiya, A. Kanade, and R. Majumdar. Race detection for Android applications. In *ACM SIGPLAN Notices*, volume 49, pages 316–325. ACM, 2014.

[72] R. Majumdar and K. Sen. Hybrid concolic testing. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 416–426. IEEE, 2007.

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

[74] S. McMaster and A. Memon. Call-stack coverage for GUI test suite reduction. *IEEE Transactions on Software Engineering*, 34(1):99–115, 2008.

[75] K. Meinke and M. A. Sindhu. Incremental learning-based testing for reactive systems. In *TAP*, pages 134–151, 2011.

[76] K. Meinke and N. Walkinshaw. Model-based testing and model inference. In *ISoLA (1)*, pages 440–443, 2012.

[77] A. Memon, I. Banerjee, and A. Nagarajan. What test oracle should I use for effective GUI testing? In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 164–173. IEEE, 2003.

[78] A. M. Memon. An event-flow model of GUI-based applications for testing. *Softw. Test., Verif. Reliab.*, 17:137–157, 2007.

[79] P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu. An overview of the MOP runtime verification framework. *International Journal on Software Techniques for Technology Transfer*, 14(3):249–289, June 2011.

[80] A. Mesbah, A. van Deursen, and S. Lenselink. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. *TWEB*, 6(1):3, 2012.

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

[82] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek. Reducing combinatorics in gui testing of Android applications. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016.

[83] N. Mirzaei, S. Malek, C. S. Pasareanu, N. Esfahani, and R. Mahmood. Testing Android apps through symbolic execution. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–5, 2012.

[84] G. Misherghi and Z. Su. HDD: hierarchical delta debugging. In *Proceedings of the 28th international conference on Software engineering*, pages 142–151. ACM, 2006.

[85] J. Nevo and P. Crégut. ASMDEX. `http://asm.ow2.org/asmdex-index.html`, 2012.

[86] W. M. Newman. A system for interactive graphical programming. In *Proc. of the spring joint computer conference (AFIPS '68 (Spring))*, pages 47–54. ACM, 1968.

[87] C. D. Nguyen, A. Marchetto, and P. Tonella. Combining model-based and combinatorial testing for effective test case generation. In *ISSTA*, pages 100–110, 2012.

[88] D. Peled, M. Y. Vardi, and M. Yannakakis. Black box checking. In *FORTE*, pages 225–240, 1999.

[89] M. Pradel, P. Schuh, G. Necula, and K. Sen. EventBreak: Analyzing the responsiveness of user interfaces through performance-guided test generation. In *ACM SIGPLAN Notices*, volume 49, pages 33–47. ACM, 2014.

[90] Z. Qin, Y. Tang, E. Novak, and Q. Li. Mobiplay: A remote execution based record-and-replay tool for mobile applications. In *Proceedings of the 38th International Conference on Software Engineering*, pages 571–582. ACM, 2016.

[91] H. Raffelt, B. Steffen, and T. Margaria. Dynamic testing via automata learning. In *Haifa Verification Conference*, pages 136–152, 2007.

[92] S. Rasthofer, S. Arzt, S. Triller, and M. Pradel. Making malory behave maliciously: Targeted fuzzing of Android execution environments. In *Proceedings of the 39th International Conference on Software Engineering. ACM. To appear*, 2017.

[93] V. Rastogi, Y. Chen, and W. Enck. AppsPlayground: automatic security analysis of smartphone applications. In *CODASPY*, pages 209–220, 2013.

[94] L. Ravindranath, S. Nath, J. Padhye, and H. Balakrishnan. Automatic and scalable fault detection for mobile applications. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pages 190–203. ACM, 2014.

[95] R. Reda. Robotium, User scenario testing for Android. `https://github.com/RobotiumTech/robotium`, 2010. Accessed: 2017-04-11.

[96] H. Reza, S. Endapally, and E. Grant. A model-based approach for testing GUI using hierarchical predicate transition nets. In *International Conference on Information Technology (ITNG'07)*, pages 366–370. IEEE Computer Society, 2007.

[97] D. J. Richardson. TAOS: Testing with analysis and oracle support. In *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, pages 138–153. ACM, 1994.

[98] R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences (extended abstract). In *STOC*, pages 411–420, 1989.

[99] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough. Precimonious: Tuning assistant for floating-point precision. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 27. ACM, 2013.

[100] C. Scott, A. Wundsam, B. Raghavan, A. Panda, A. Or, J. Lai, E. Huang, Z. Liu, A. El-Hassany, S. Whitlock, et al. Troubleshooting blackbox SDN control software with minimal causal sequences. *ACM SIGCOMM Computer Communication Review*, 44(4):395–406, 2015.

[101] M. Shahbaz and R. Groz. Inferring mealy machines. In *FM*, pages 207–222, 2009.

[102] C. Sharma, S. Sabharwal, and R. Sibal. A survey on software testing techniques using genetic algorithm. *arXiv preprint arXiv:1411.1154*, 2014.

[103] R. K. Shehady and D. P. Siewiorek. A methodology to automate user interface testing using variable finite state machines. In *FTCS*, pages 80–88, 1997.

[104] A. Shi, A. Gyori, M. Gligoric, A. Zaytsev, and D. Marinov. Balancing trade-offs in test-suite reduction. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 246–256. ACM, 2014.

[105] A. Shi, T. Yung, A. Gyori, and D. Marinov. Comparing and combining test-suite reduction and regression test selection. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 237–247. ACM, 2015.

[106] G. Shu and D. Lee. Testing security properties of protocol implementations-a machine learning based approach. In *Distributed Computing Systems, 2007. ICDCS'07. 27th International Conference on*, pages 25–25. IEEE, 2007.

[107] StackOverFlow. Stack Overflow: what would be the base optimal throttle and seed for an application using monkey test? `http://stackoverflow.com/questions/9778881`, 2014. Accessed: 2017-04-14.

[108] T. Takala, M. Katara, and J. Harty. Experiences of system-level model-based GUI testing of an Android application. In *ICST*, pages 377–386, 2011.

[109] F. Vaandrager. Model learning. *Communications of the ACM*, 60(2):86–95, 2017.

[110] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot-a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[111] N. Walkinshaw, K. Bogdanov, J. Derrick, and J. Paris. Increasing functional coverage by inductive testing: A case study. In *ICTSS*, pages 126–141, 2010.

[112] N. Walkinshaw, J. Derrick, and Q. Guo. Iterative refinement of reverse-engineered models by model-based testing. In *FM*, pages 305–320, 2009.

[113] L. White and H. Almezen. Generating test cases for GUI responsibilities using complete interaction sequences. In *11th International Symposium on Software Reliability Engineering (ISSRE'00)*, page 110. IEEE Computer Society, 2000.

[114] T. Xie and D. Notkin. Mutually enhancing test generation and specification inference. In *FATES*, pages 60–69, 2003.

[115] X. Yan, J. Han, and R. Afshar. Clospan: Mining: Closed sequential patterns in large datasets. In *Proceedings of the 2003 SIAM International Conference on Data Mining*, pages 166–177. SIAM, 2003.

[116] S. Yang, D. Yan, and A. Rountev. Testing for poor responsiveness in Android applications. In *Engineering of Mobile-Enabled Systems (MOBS), 2013 1st International Workshop on the*, pages 1–6. IEEE, 2013.

[117] W. Yang, M. R. Prasad, and T. Xie. A grey-box approach for automated GUI-model generation of mobile applications. In *FASE*, pages 250–265, 2013.

[118] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

[119] X. Yuan, M. Cohen, and A. M. Memon. Covering array sampling of input event sequences for automated GUI testing. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 405–408, New York, NY, USA, 2007. ACM.

[120] X. Yuan and A. M. Memon. Iterative execution-feedback model-directed GUI testing. *Information & Software Technology*, 52(5):559–575, 2010.

[121] R. N. Zaeem, M. R. Prasad, and S. Khurshid. Automated generation of oracles for testing user-interaction features of mobile apps. In *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*, pages 183–192. IEEE, 2014.

[122] A. Zeller. Yesterday, my program worked. Today, it does not. Why? In *Software EngineeringESEC/FSE99*, pages 253–267. Springer, 1999.

[123] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 1–10. ACM, 2002.

[124] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[125] H. Zhang, H. Wu, and A. Rountev. Automated test generation for detection of leaks in Android applications. In *Automation of Software Test (AST), 2016 IEEE/ACM 11th International Workshop in*, pages 64–70. IEEE, 2016.

[126] H. Zhong, L. Zhang, and H. Mei. An experimental study of four typical test suite reduction techniques. *Information and Software Technology*, 50(6):534–546, 2008.

[127] E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach. *IEEE transactions on Evolutionary Computation*, 3(4):257–271, 1999.