USING MACHINE LEARNING TO IMPROVE AUTOMATED TEST GENERATION

by

Yavuz Köroğlu

B.S., Computer Engineering, Boğaziçi University, 2014

M.S., Computer Engineering, Boğaziçi University, 2016

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

Graduate Program in Computer Engineering

Boğaziçi University

2022

ACKNOWLEDGEMENTS

First, I would like to thank my supervisor Prof. Alper Şen for guiding me with his experience and skill, and fellow jury members Prof. Can Özturan, Assoc. Prof. Emre Uğur, Assoc. Prof. Hasan Sözer, and Assoc. Prof. Feza Buzluca for their invaluable feedback.

I would like to thank especially Dilara Nejad, my fiancée, for I would not find the strength to finish this thesis without her.

This author was supported by TUBITAK Ph.D. Fellowship 2011-E and supported in part by Bogazici University Research Fund 13662.

ABSTRACT

USING MACHINE LEARNING TO IMPROVE AUTOMATED TEST GENERATION

Underestimating the value of software testing had catastrophic results in recent history. Automated Test Generation (ATG) is an approach that aims to minimize the manual effort required for testing. This thesis aims to improve the effectiveness and performance of ATG approaches via Machine Learning (ML) based guidance, and focuses on Android Graphical User Interface (GUI) testing using Reinforcement Learning (RL), specifically. We propose four solutions, Q-learning Based Exploration (QBE), Test Case Mutation (TCM), Fully Automated Reinforcement LEArning Driven (FARLEAD), and FARLEAD2 test generators. QBE uses RL to crawl a set of applications and learns an action generation policy while exploring. Then, it uses this learned policy to either detect more unique crashes or cover more activities in new applications. TCM takes the tests QBE generates and replaces the well-behaving actions in those tests with bad-behaving ones to detect even more crashes. FARLEAD uses RL to learn how to verify a functional behavior that is given as a high-level test scenario in the form of a monitorable formal specification. FARLEAD learns by trial-and-error like QBE but it learns app-specific patterns instead of QBE's app-generic patterns. To the best of out knowledge, FARLEAD is the first engine fully automating the functional testing of GUI applications. Finally, FARLEAD2 improves FARLEAD with Generalized Experience Replay (GER) and human-readable Staged Test Scenario (STS) language. Experimental results show that, QBE outperforms state-of-the-art test generators in crash detection and coverage. Furthermore, executing QBE first and then switching to TCM detects even more unique crashes. FARLEAD and FARLEAD2 expand the scope of automated testing to verifying functional behavior. Overall, these test generators elevate automated GUI testing closer to replacing manual GUI testing.

ÖZET

OTOMATİK TEST YARATIMINI İYİLEŞTİRME AMAÇLI MAKİNE ÖĞRENMESİ KULLANIMI

Yazılım testinin değerini hafife almanın yakın tarihte yıkıcı sonuçları olmuştur. Otomatik Test Yaratımı (OTY) test için gereken insan eforunun en aza indirilmesini amaçlayan bir yaklaşımdır. Bu tez OTY etkililiği ve performansını Makine Öğrenmesi (MÖ) tabanlı yönlendirme ile artırmayı amaçlamaktadır, ve spesifik olarak Takviyeli Öğrenme (TÖ) kullanan Android Grafiksel Kullanıcı Arayüzü (GKA) testine odaklanmaktadır. Önerdiğimiz dört çözüm; Q-öğrenme Tabanlı Keşif (QTK), Test Durumu Mutasyonu (TDM), Tam Otomatik Takviyeli Öğrenme Güdümlü (TOTÖG), ve TOTÖG2 test yaratıcılarıdır. QTK bir dizi uygulamada TÖ kullanarak gezinir ve keşif sırasında bir eylem yaratma ilkesi öğrenir. Sonra, bu öğrendiği ilkeyi yeni uygulamalarda ya daha fazla özgün çökme bulmak ya da daha fazla aktivite kapsamak için kullanır. TDM, QTK ile yaratılan testleri alır ve içlerindeki iyi huylu eylemleri daha da fazla çökme tespit edebilmek için kötü huylularla değiştirir. TOTÖG ise TÖ kullanarak izlenebilir kurallı belirtimler formundaki yüksek seviyeli test senaryoları olarak verilen fonksiyonel davranışları nasıl doğrulayacağını öğrenir. TOTÖG, QTK gibi deneme-yanılma ile öğrenir ama uygulama-spesifik kalıplar öğrenmektedir. Bildiğimiz kadarıyla, TOTÖG, GKA uygulamalarının tam otomatik fonksiyonel testini mümkün kılan ilk motordur. Son olarak, TOTÖG2, TOTÖG'ü Genellenmiş Deneyim Tekrarı (GDT) ve insan-okuyabilir Aşamalı Test Senaryosu (ATS) diliyle geliştirmektedir. Deneyler QTK'nın en gelişkin test yaratıcılarından çökme tespiti ve kapsamada daha performanslı olduğunu göstermektedir. Önce QTK çalıştırıp sonra TDM'ye geçmek ise bundan da daha fazla eşsiz çökme bulmaktadır. TOTÖG, otomatik testin kapsamını fonksiyonel davranış doğrulanmasına genişletmektedir. Sonuç olarak, bu test yaratıcıları otomatik GKA testini elle testin yerine geçmeye yakınlaştırmaktadır.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

t	A test
v	A GUI state
s	A GUI-monitor state
TS	Test suite
δ	A Mutation Operator
Δ	Mutation Operator Space
$\neg$	Negation
$\top$	True
$\neg\top$	False
$\wedge$	Conjunction
$\vee$	Disjunction
$\bigcirc$	Next
$\mathcal{U}$	Until
$\Diamond$	Eventually
$\Box$	Globally

LIST OF ACRONYMS/ABBREVIATIONS

ARB	Aging Related Bug
ATG	Automated Test Generation
AUT	Application Under Test
ELTS	Extended Labeled Transition System
ER	Experience Replay
FARLEAD	Fully Automated Reinforcement LEArning Driven
GER	Generalized Experience Replay
GUI	Graphical User Interface
LTL	Linear-time Temporal Logic
MATE	Mobile Accessibility Testing
ML	Machine Learning
NAM	Non-Aging Related Mandelbug
PUMA	Programmable UI-automation of Mobile Apps
QBE	Q-learning Based Exploration
RE	Random Exploration
RL	Reinforcemenet Learning
RND	RaNDom
STS	Staged Test Scenario
TCM	Test Case Mutation

1. INTRODUCTION

Testing ensures high quality and reliability in software. Costs of inadequate testing can be catastrophic. For example, Knight Capital Group lost $440M dollars due to a bug in their trading software in 2012 [1]. One of the first computer worms, the *Morris Worm*, was developed in 1988 and exploited several holes in the Unix operating system [2]. United States Government Accountability Office announced that these bugs caused a huge damage, costing between $100K and $10M dollars. A division by zero bug in early Intel Pentium processors cost $475M dollars in 1994 [3]. A study conducted by National Institute of Standards and Technology (NIST) in 2002 reports that software bugs cost the U.S. economy $59.5B dollars annually [4].

Adequate testing requires time and a lot of effort. Testing with manually generated inputs is the predominant technique in industry to ensure software quality. This type of testing accounts for up to 80% of the typical cost of software development (e.g. in Microsoft, 79% of developers are dedicated to writing unit tests [5]), but manual test generation is expensive, error-prone, and rarely exhaustive.

Several techniques under the name of Automated Test Generation (ATG) have been proposed to automatically generate test inputs [6]. Research on ATG dates back to 1975 [7] and although there are many scientific works on the subject, real-world applications are limited. Our research aims to find the challenges to bridge the gap between the scientific research and the real-world application and improve the scientific research to better fit the real-world requirements. Note that ATG is different than Test Automation. In Test Automation, the focus is on libraries that help generating tests manually. In ATG, tests are generated, executed and reported automatically with no human intervention.

We started our research on ATG by first looking into mobile applications because mobile applications are ubiquitous, in other words, accessible and used by everyone.

After 2020, there are over 30 billion smart mobile devices worldwide and this number is expected to go up to 75 billion until 2025 [8].

Graphical User Interface (GUI) applications play a huge role in mobile devices. Naturally, many GUI testing studies focus on mobile GUI applications. Statistics show that between 2019 and 2021, on average, a hundred thousand new mobile applications have been released on Google Play every month [9]. Reducing the time needed for adequate testing of these applications is critical to ensure correct behavior while keeping the rate of releases high.

A bibliometric analysis [10] shows that the study of automated GUI testing has been continuously growing for the last 30 years. Automated GUI testing aims to reduce the manual effort and time spent on testing. A comparative study on automated testers [11] shows that black-box methods are the least time-consuming compared to other alternatives. Considering the scarcity of time to test, black-box methods should be preferable in GUI testing. Note that black-box methods work only on the executable binary of an Application Under Test (AUT) and do not need the source code. However, the same study also reveals that black-box methods often focus on external exceptions and ignore functional behavior. In GUI testing, the developer/tester must ensure the correct behavior, so automating the functional testing of GUI applications is essential to reduce the manual effort and time spent on testing.

To the best of our knowledge, all scientific papers about Mobile GUI Testing in the literature study Android GUI and not the other Mobile GUIs. This has several reasons.

(i) Android applications have the largest share in the mobile application market, where 82.8% of all mobile applications are designed for Android [8].
(ii) There are large databases of freely downloadable Android applications. F-Droid [12] is one such database which consists 4521 Android applications.
(iii) Android OS is open source, which makes it easy to modify the OS to monitor the

application state.

(iv) Android SDK is freely accessible unlike SDKs for other mobile platforms such as iOS.

Android GUI application developers face a pressure to develop many applications in limited time. This pressure combined with the lack of test automation tools cause most applications in the market to be buggy. There are many test generators for Android GUIs [13–18]. These tools are fully-automated, in other words, they generate and execute tests and produce reports for the developer without any human intervention. We observed that these tools cannot achieve high coverage and cannot detect a high number of crashes. In fact, previous testing tools fail to outperform a simple random test generator called Monkey [14, 17].

In this thesis, we first propose the usage of machine learning to improve testing. We hypothesize that in the Android GUI Testing domain there are general patterns which can be learned from a training set of applications and then used on new applications with improvement in crash detection and coverage. We applied a known Reinforcement Learning (RL) technique called SARSA, which is an on-policy version of Q-Learning on a training set of 200 applications, downloaded from the F-Droid website. Then, we evaluated our approach by executing a test set of 100 applications, again downloaded from the F-Droid benchmarks, and compared our results with the previous tools. We published our work as *QBE: QLearning-Based Exploration of Android Applications* [19] at 2018 IEEE International Conference on Software Testing, Validation, and Verification (ICST).

Second, we propose the usage of Test Case Mutation (TCM) to improve crash detection. After QBE, we observed that there are still many crashes that were not detected. We investigated Android crash patterns described in the literature [20] to uncover the methods to hit these crashes in our tests. We worked on five case studies to relate the crash patterns with specific actions and developed six mutation operators. These operators take the existing test cases and inject or modify specific actions in

the test case to produce a bad-behaving test case that is related to the Android crash patterns. We published our work as *TCM: Test Case Mutation to Improve Crash Detection in Android* [21] at 2018 International Conference on Fundamental Approaches to Software Engineering (FASE).

Third, we notice that all Android GUI test generators focus on crash/bug detection and structural coverage but the main goal of GUI testing is to verify functional behavior. We propose the Fully Automated Reinforcement LEArning-Driven test generator (FARLEAD) that uses RL to fully automate functional testing of Android GUI applications. Since FARLEAD aims to test application-specific functions, it becomes unreasonable to investigate general patterns. Instead, FARLEAD takes a test scenario, which is an example use case of a GUI function. Note that FARLEAD accepts Linear-time Temporal Logic (LTL) specifications as test scenarios because LTL formulae are unambiguously run-time monitorable. By trial-and-error, FARLEAD learns to generate the correct low-level GUI actions in the correct order such that the resulting test witnesses a given test scenario, which verifies the functional behavior of the Application Under Test (AUT). We published our work as *Functional test generation from UI test scenarios using reinforcement learning for android applications* [22].

Finally, our experience with FARLEAD [23] shows that it may still be impractical for large-scale real-world use due to two reasons: (i) FARLEAD witnesses simple test scenarios but as the test scenario gets more complex, it fails to reach functional behavior located deep in the AUT, and (ii) writing and maintaining test scenarios in a formal specification language such as LTL remains a difficult task for the developer/tester. For this reason, we propose FARLEAD2, an improvement to FARLEAD. We improve FARLEAD by enhancing its Reinforcement Learning (RL) algorithm with Generalized Experience Replay (GER). GER gathers experience while witnessing a test scenario and then utilizes that experience for later test scenarios. We also propose the Staged Test Scenario (STS), a test scenario type that divides the underlying tasks of a test scenario into consecutive stages. These stages produce intermediate positive rewards upon completion, enabling Reward Shaping. Note that Reward Shaping [24] is the

practice of generating intermediate rewards before reaching an objective to drive the RL agent towards that objective. STS is an unambiguous and run-time monitorable but still human-readable test scenario language with intuitive syntax and semantics. STSs allow the developer/tester to easily incorporate apriori information on the AUT, facilitating witness generation.

1.1. Contributions

Our key contributions in this thesis are as follows.

- We propose Q-learning Based Exploration (QBE) that implements a popular semi-supervised Machine Learning (ML) technique called Reinforcement Learning (RL) to improve coverage and crash detection of automated test generation tools.
- We propose Test Case Mutation (TCM) that mutates test inputs QBE generates to further improve crash detection of automated test generation tools.
- An experimental environment for QBE/TCM which consists of
  - 4521 Android applications downloaded and instrumented for collecting coverage from the F-Droid benchmarks,
  - 7 identical real-world Android devices,
  - 14 Android-x86 VirtualBox guests,
  - An environment that allows parallel execution of different techniques on different applications simultaneously.
- Experiments involving 200 random applications as training set and 100 random applications as the test set. Experiments show that QBE/TCM indeed improves coverage and crash detection.
- We propose FARLEAD, the first fully automated mobile GUI test generator for functional testing that uses RL.
- We evaluate FARLEAD via experiments on two applications from F-Droid. We show that our approach is more effective and achieves higher performance in generating satisfying tests than three known test generation approaches, namely Monkey, Random, and QBEa.

- We propose FARLEAD2, the first study that combines RL and Generalized Experience Replay (GER) for GUI test generation,
- We develop the Staged Test Scenario (STS) language to utilize readable and unambiguously monitorable test scenarios.
- We made an experimental evaluation of RL with GER, showing that it fails fewer times, witnesses more test scenarios, and is faster than RaNDom (RND) and pure RL witness generators.

1.2. Organization

We organized this thesis as follows. We provide literature review in Chapter 2. We describe the necessary background on Android GUI Testing and Reinforcement Learning in Chapter 3. We describe our methods in Chapters 4-7. We discuss the limitations of our test generators in Chapter 8. We conclude by making a short summary of this thesis and stating future avenues of research in Chapter 9.

2. RELATED WORK

In this section, we discuss the published literature related to our work in five categories:

(i) Android Test Generators in Section 2.1,
(ii) Other GUI Test Generators in Section 2.2,
(iii) Record and Replay in Section 2.3,
(iv) Runtime Verifiers in Section 2.4, and
(v) RL-LTL Studies in Section 2.5.

2.1. Android Test Generators

There are many test generators for Android mentioned in the published literature since both Android systems are widespread and Automated Test Generation receives some academic focus. Table 2.1 shows 23 automated test generators we investigated, in chronological order. Although we cannot put a year on Google's Monkey testing tool, it is the oldest tool shipped in bundle with every Android OS version.

Black-box test generators are more practical than others because they do not need the AUT's source code. Table 2.1 shows that there are several black-box test generators. All the test generators we implement during this thesis are also black-box.

Most Android Test Generators in the literature focus on crash detection and/or structural coverage. A crash in Android is an abnormal termination of the Application Under Test (AUT) followed by a *FATAL EXCEPTION* written in Android logs. Note that a crash is a bug but a bug may not be crash. For example, if a banking application executes a transfer without any crashes but sends a wrong amount of money, that is a non-crashing bug. Structural coverage is a measure of how much of the AUT a test suite explores. A typical structural coverage metric could be line, branch, or

Table 2.1. Android Test Generators.

Year	#	**Name**	**Black-box**	**Goal**
N/A	1	Monkey [25]	✔	Crash Detection
2012	2	ACTEve [26]	✖	Structural Coverage
2013	3	A^{3}E [14]	✔	Structural Coverage
	4	DynoDroid [18]	✖*	Coverage & Crash
	5	Orbit [27]	✖	Structural Coverage
	6	SwiftHand [15]	✔	Structural Coverage
2014	7	EvoDroid [28]	✖	Structural Coverage
	8	GreenDroid [29]	✖	Energy
	9	MobiGUITAR [30]	✔	Crash Detection
	10	PUMA [13]	✔	Coverage & Crash
	11	Quantum [31]	✖	Common Bugs
2015	12	MonkeyLab [32]	✖	Structural Coverage
2016	13	CrashScope [17]	✔	Crash Detection
	14	Sapienz [16]	✔	Coverage & Crash
	15	TrimDroid [33]	✖	Structural Coverage
2017	16	DroidBot [34]	✔	Sensitive APIs
	17	Stoat [35]	✔	Structural Coverage
2018	18	CrawlDroid [36]	✔	Crash Detection
	19	MATE [37]	✔	Accesibility
	20	SwiftHand2 [38]	✔	Structural Coverage
	21	LAND [39]	✔	Structural Coverage
2019	22	Paraaim [40]	✔	Structural Coverage
2021	23	GENIE [41]	✔	Common Bugs

*DynoDroid also instruments the Android OS.

method coverage. In Android, it is also common to use activity coverage, which roughly measures the number of screens explored over the total number of screens available to the AUT.

Exceptions to crash detection and structural coverage focus on common or specific bugs, specific bugs being related to triggering sensitive APIs, excess energy consumption and accessibility issues. Overall, these test generators detect exceptions and achieve high structural coverage, though it is unclear how many of the GUI functions they test. In practice, a test generator may achieve high structural coverage and still fail to test essential GUI functions, missing the mark on the main goal of GUI testing, which is to verify functional behavior. Instead, our proposed test generators focus on structural coverage, crash detection, and functional testing all together.

2.2. Other GUI Test Generators

In the larger domain of general GUI testing, the idea of Reinforcement Learning (RL) is not new. To the best of our knowledge, AutoBlackTest [42] is the first study on RL-driven GUI test generation. AntQ [43] uses advanced ant colony optimization techniques along with reinforcement learning. These test generators learn app-specific patterns while exploring, aiming to explore as many widgets as possible. In this thesis, we propose several approaches, beginning from app-generic patterns in QBE to find crashes or increase coverage and then switch to app-specific learning in FARLEAD to verify functional behaviors. The advantage of using app-generic patterns is that AutoBlackTest and AntQ have to spend some time to learn every new AUT whereas once QBE learns enough, it does not need learning time for the new AUTs. Later RL driven GUI test generators, namely TESTAR [44], and others [45, 46] continue to optimize structural testing, whereas FARLEAD enabled fully automated functional testing.

2.3. Record and Replay

One intuitive way to verify a crash, reachability of a screen (activity coverage), or a functional behavior is to record a replayable test. To this end, a record and replay tool records every manual user action as test steps and then replays those steps in the recorded order. RERAN [47], VALERA [48], and BARISTA [49] are example record

and replay tools. These tools record GUI actions performed by hand and then replay the same actions to reproduce the original test. We also reproduce our results the same way, except recorded actions are not performed by hand but by the test generator.

2.4. Runtime Verifiers

A Runtime Verification (RV) tool monitors the AUT and reports if any given specification is violated or not. RV-Droid [50], RV-Android [51], ADRENALIN-RV [52], and Android-SRV [53] monitor Android AUTs given LTL specifications. However, these tools monitor LTL properties on a source code level. Instead, the monitoring we have developed during this thesis is at the GUI level. Also, runtime verification tools do not generate tests whereas our proposed tools generate relevant tests while monitoring.

2.5. RL-LTL Studies

Several studies [54–57] develop RL-LTL systems. Using RL, these systems learn to obey constraints specified in LTL language. These RL-LTL systems have to continuously perform their given task, without termination. Hence, as typical RL-LTL approaches, they must converge to an optimal policy to guarantee the highest reliability. Instead, FARLEAD is an RL-LTL system with a finite task, so it can terminate once the task is complete. Therefore, FARLEAD does not have to converge to an optimal policy, saving from the learning time. The overall result is that a typical RL-LTL system may require around 100K steps [39, 54–56] to learn whereas 1K steps is mostly enough for FARLEAD.

3. BACKGROUND

In this chapter, we provide the background necessary for the understanding of this thesis. We first provide information on Android GUIs in Section 3.1. Then, we explain Reinforcement Learning (RL) in Section 3.2. Finally, we describe our test adequacy criteria and their relation to Linear-time Temporal Logic (LTL) formulae.

3.1. Android GUI Basics

Android GUI is based on *activities*, *events*, and *crashes*. An *activity* is a container for a set of GUI widgets. These GUI widgets are visible on the Android screen. Every widget has properties describing its boundaries in pixels (x_1, y_1, x_2, y_2) or how the user can interact with it (e.g. *enabled*, *clickable*, *longclickable*, *scrollable*, *password*). We use the *type* and *password* properties to determine if a widget is something a user can write on.

The Android system and the user can interact with GUI components using *events*. We divide events in two categories, *system events* and *GUI actions*. We describe GUI actions in Section 3.1.2. We assume that after every GUI action, the Android GUI goes into what we call a GUI state, waiting for another GUI action. We discuss these GUI states in Section 3.1.1. Finally, we model an Android GUI Application as a transition system in Section 3.1.3.

3.1.1. Android GUI State

An Android GUI state is the state of an Android device between two GUI actions. The contents of a GUI state depends on how fine-grained states the underlying test generator needs. Test generators that rely on learning a policy of making similar actions on similar states relax or abstract out the definition of a state, so similar states count as the same. Otherwise, the definition of a state is as fine-grained as possible. The

most fine-grained definition of a state consists four components:

(i) The package name of the AUT,
(ii) The activity name of the current screen,
(iii) Contextual attributes (Crashed, Wi-Fi, Bluetooth, GPS, etc.), and
(iv) The widget tree.

Baek and Bae [58] define five levels of comparison between states, ignoring how it is defined but grouping similar states together. The maximum comparison level requires two states to be exactly equal according to its finest-grained definition.

A state is *crashed* if a fatal exception is recorded in Android logs [17,20]. Crashes often result with the AUT terminating with or without any warning. Some crashes do not visually affect the execution, but the AUT halts as a result.

3.1.2. Android GUI Action

A GUI action is based on user gestures interacting with the Android GUI. Table 3.1 shows the GUI actions we support. The first 10 actions are non-contextual GUI actions more or less supported by any test generator. Connectivity, Bluetooth, Location, Planemode, and Doze change the contextual attributes of a state, so they are contextual actions. These are not simple GUI actions, typically performing them require multiple clicks. However, our test generators perform these actions in one go by triggering system events to change contextual attributes. Finally, Reinit is a special action that restarts the AUT.

Menu, Back, 2×Back, and the contextual actions are universal actions that are always enabled. Click, Long-Click, Scroll-Up, Scroll-Down, Scroll-Left, Scroll-Right, and Write have related GUI widgets, where a GUI widget is a GUI component visible on the screen. These actions are enabled only if a related GUI widget appears on the screen. Technically, a test generator determines the set of currently enabled actions by

Table 3.1. GUI Actions.

	Action Type	Universal	Related Widget	Parameters
Non-Contextual	Menu	✔	✖	-
	Back	✔	✖	-
	2×Back	✔	✖	-
	Click	✖	✔	-
	Long-Click	✖	✔	-
	Scroll-Up	✖	✔	-
	Scroll-Down	✖	✔	-
	Scroll-Left	✖	✔	-
	Scroll-Right	✖	✔	-
	Write	✖	✔	text
Contextual	Connectivity	✔	✖	on/off/toggle
	Bluetooth	✔	✖	on/off/toggle
	Location	✔	✖	gps/gps&network/off/toggle
	Planemode	✔	✖	on/off/toggle
	Doze	✔	✖	on/off/toggle
Sp	Reinit	✖	✖	-

parsing the XML hierarchy of the widget tree. Only the Write action has a parameter, which is what text to write on its related GUI widget. In our early studies, we take this parameter from a dictionary. But later, with the introduction of test scenarios, we also started to deduce it from the test scenario.

3.1.3. Extended Labeled Transition System

Test generators model the Android GUI as a transition system where states are the nodes and actions are the edges connecting them. *Extended Labeled Transition System* (ELTS) [15] is a known construct that models the AUT. Formally, an ELTS $M = (S, s_0, Z, \omega, \lambda)$ is a 5-tuple, where

- S is a set of *states* (vertices),
- $s_0 \in S$ is the *initial state*,
- Z is the set of all *actions* (input alphabet),

- $\omega : S \times S \times Z$ is the *state transition relation*, and
- $\lambda : S \to \wp(Z)$ is a *state labeling function*, where $\forall s \in S, \lambda(s) \subseteq Z$ denotes the set of actions enabled at state s.

ELTS extends a typical finite state transition system by labeling every state. SwiftHand [15] uses these labels for comparing states and picking a valid action at every state. Later, with the introduction of test scenarios, we replace these labels with Boolean propositions coming from the test scenario. Note that this modification allows the same GUI state having different labels, depending on the previous actions of a test. Hence, λ becomes not just a function of the GUI state, but also the monitor state of the test scenario.

A *transition* in an ELTS is a triple, (s, s', z). TCM mutates the time spent on a transition, so define a *delayed transition* as a quadruple, (start-state, end-state, action, delay in seconds), shortly (s, s', z, d). A test t is a sequence of transitions (or delayed transitions in the case of TCM), starting with the initial state s_0. A *test suite* ts is a set of tests.

3.2. Reinforcement Learning

Reinforcement Learning (RL) is a semi-supervised machine learning methodology. It has lead to impressive advances in artificial intelligence, exceeding human performance in areas including but not limited to resource management [59], traffic light control [60], playing sophisticated games such as chess [61] and atari [62], and chemistry [63]. Figure 3.1 shows that an RL agent dynamically learns to perform its task by trial-and-error. After every action, the RL agent receives an immediate reward from the environment. This reward can be positive, negative, or zero, meaning that the last decision of the RL agent was good, bad, or neutral, respectively. Decisions made according to the RL agent's experience is said to follow the agent's policy. After enough iterations, the RL agent becomes proficient in its task. At this point, the RL agent is said to have converged to its optimal policy. Upon convergence, the RL agent

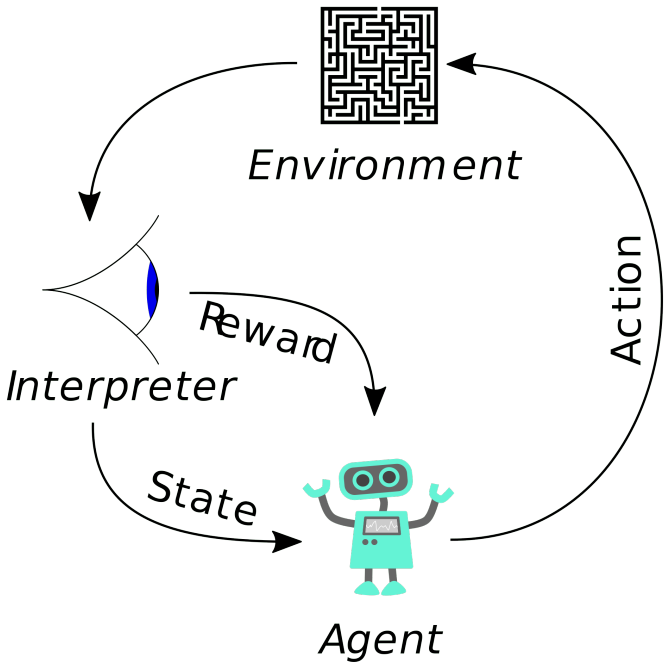


Figure 3.1. Reinforcement Learning Overview.

is said to have minimized its expected number of bad decisions in the future. The RL agent requires no prepared training data, which decreases the manual effort spent preparing it. Therefore, RL is attractive amongst many machine learning methods.

Test generation is a *search problem*. Therefore, we could use any search algorithm as an alternative to RL to generate candidate tests. For example, in Android GUI Testing, EvoDroid [28] and Sapienz [16] use evolutionary search algorithms instead of RL.

In this thesis, we use RL instead of evolutionary search because

(i) From Sutton and Barto [64, p.8], we know that the evolutionary search ignores the fact that the states and actions are known, so it yields high execution costs. We avoid these costs by prefering RL over evolutionary search. In contrast to evolutionary search, RL learns while interacting with the environment and takes individual behavioral interactions into account, which is more efficient than evolutionary search in most cases.

(ii) The main difference between RL and evolutionary search is the *adaptation* scheme [65]. In every episode, the evolutionary search generates a candidate population and updates it by replacing bad candidates with better ones according to a fitness function. Instead, in every episode, RL produces only one individual candidate and updates its candidate generation policy after every step according to a reward function. Our focus is to generate just one test for a specific UI test scenario, and executing candidate tests as in evolutionary search is costly. So we prefer RL because it has the potential to find the witness before learning the optimal policy and it avoids generating large numbers of candidates as in evolutionary search.

3.2.1. Update Equations

Table 3.2. Known Update Equations.

TD-Learning	$\delta = r + \gamma V(s_{k+1}) - V(s)$
Q-Learning	$\delta = r + \gamma \max_a Q(s_{k+1}, a) - Q(s_k, a_k)$
SARSA	$\delta = r + \gamma Q(s_{k+1}, a_{k+1}) - Q(s_k, a_k)$
Expected SARSA	$\delta = r + \gamma \sum_a P_{pol}(a\|s_{k+1}) Q(s_{k+1}, a) - Q(s_k, a_k)$
Double TD-Learning	$\delta = r + \gamma V_2(s_{k+1}) - V_1(s)$
Double Q-Learning	$\delta = r + \gamma \max_a Q_2(s_{k+1}, a) - Q_1(s_k, a_k)$
Double SARSA	$\delta = r + \gamma Q_2(s_{k+1}, a_{k+1}) - Q_1(s_k, a_k)$
Expected Double SARSA	$\delta = r + \gamma \sum_a P_{pol}(a\|s_{k+1}) Q_2(s_{k+1}, a) - Q_1(s_k, a_k)$

Table 3.2 shows the well-known update equations for RL. At every k^{th} step, RL calculates the update amount δ using one of these equations, where r denotes the immediate reward, Q and V denote the expected future reward of a state-action pair and a state, respectively, and finally, P_{pol} denotes the probability of chosing an action a given a next state s_{k+1} according to a learned policy *pol*.

In our thesis work, we use a simplification of all Q-matrix based equations in Table 3.2 called Myopic Updates as

$$\delta = r - Q(s, a) \tag{3.1}$$

We discard the contribution of future states during updates by taking the discount factor λ as zero. Hence only the immediate reward r and the old Q-value $Q(s, a)$

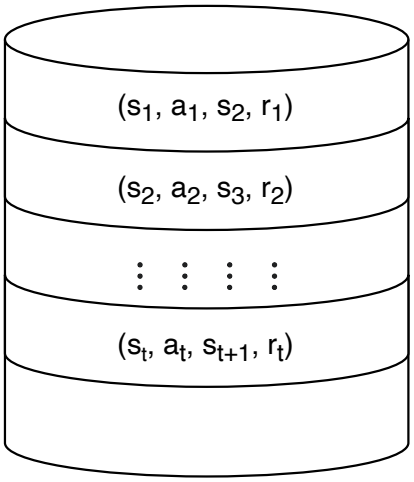


Figure 3.2. Experience Database.

remain relevant to the updates. We let the future rewards affect the Q-matrix via backpropagating future Q-values on the state-action pairs of the latest execution trace. This trace is also called Eligibility Trace e.

3.2.2. Experience Replay

Experience Replay (ER) [66] improves RL by using the experience collected in an Experience Database from previous tasks instead of throwing it away. Figure 3.2 shows an Experience Database as a list of unit experiences. A unit experience is a quadruple (s, a, s', r), meaning that the AUT goes from one state s to state s' by executing action a, getting reward r in the process. We never execute the unit experiences on the environment since rewards are readily available. Instead, we update the RL's initial policy according to these rewards, allowing the RL agent to start with an initially better action generation policy than a random one. So, the RL agent converges faster while avoiding the execution costs of all the unit experiences. More unit experiences result in faster learning. Note that introducing test scenarios make their monitor states relevant to any experience. Hence, the state s in an Experiene Database is the concatenation of the device state and the monitor state.

3.3. Test Scenarios and Linear-time Temporal Logic

A test scenario is an example use case of a software requirement, verifying a function's behavior if a test witnesses it during execution. A test scenario is typically given in natural language and therefore ambiguous. Test automation tools require test implementation to disambiguate the semantics of a test scenario. However, implementing extra code for a test is costly, impractical, and may introduce new bugs.

The reason for the large body of previous work is limited to bugs and coverage is that such criteria have well-known automated test oracles. On the other hand, to the best of our knowledge, a general automated oracle does not exist for functional behavior. Such an automated oracle may be borrowed from the domain of formal verification in the form of a co-safe Linear-time Temporal Logic (LTL) specification.

We define the syntax of an LTL formula ϕ where $p \in \mathcal{AP}$ is an atomic proposition where $\mathcal{AP}$ is the set of all atomic propositions, as

$$\phi := \top | p | \neg\phi | \phi \wedge \phi | \bigcirc\phi | \phi \mathcal{U} \phi \tag{3.2}$$

We interpret ϕ over a test t using the finite and pointwise semantics in Table 3.3(a).

Table 3.3. Pointwise and Finite LTL Semantics.

(a) Core Definitions.

(t,k)	$\models \top$		
(t,k)	$\models p$	iff	$p \in L(a_k) \cup L(s_k)$
(t,k)	$\models \neg\phi$	iff	$(t,k) \not\models \phi$
(t,k)	$\models \phi \wedge \phi'$	iff	$(t,k) \models \phi$ and $(t,k) \models \phi'$
(t,k)	$\models \bigcirc\phi$	iff	$(t,k+1) \models \phi$
(t,k)	$\models \phi \mathcal{U} \phi'$	iff	$\exists j \in \mathbb{N}, k \leq j < \lvert t \rvert, (t,j) \models \phi'$,
		and	$\forall i \in \mathbb{N}, [k \leq i < j \rightarrow (t,i) \models \phi]$
t	$\models \phi$	iff	$(t,0) \models \phi$

(b) Additional Definitions.

$\phi \vee \phi'$	$\triangleq$	$\neg(\neg\phi \wedge \neg\phi')$
$\phi \rightarrow \phi'$	$\triangleq$	$\neg(\phi \wedge \neg\phi')$
$\phi \leftrightarrow \phi'$	$\triangleq$	$\neg(\phi \wedge \neg\phi') \wedge \neg(\neg\phi \wedge \phi')$
$\Diamond\phi$	$\triangleq$	$\top \mathcal{U} \phi$
$\Box\phi$	$\triangleq$	$\neg[\top \mathcal{U} \neg\phi]$

We interpret ϕ over a (finite) trace $t = a_0 s_0 a_1 s_1 \ldots a_{|t|-1} s_{|t|-1}$ using the finite and pointwise semantics in Table 3.3(a). We first define that $(t,k) \models \top$ because every trace t, at any step k, entails true, denoted by $\top$. Second, we define that $(t,k) \models p$ if and only if the atomic proposition p is a label of the k^{th} state or the k^{th} action in the trace t. Third and fourth, we define the unary negation ($\neg$) and the binary conjunction

($\wedge$) operators for LTL, respectively. Fifth, we define the temporal next operator ($\bigcirc$), where a trace t satisfies $\bigcirc\phi$ at a step k if and only if it satisfies ϕ at the subsequent step $k+1$. Sixth, we define the temporal until operator ($\mathcal{U}$), where a trace t satisfies $\phi\mathcal{U}\phi'$ if and only if there exists a future step j in the (finite) trace at which the trace t satisfies ϕ', and in all steps from k up to j, the trace t satisfies ϕ. Finally, we define that a trace t satisfies an LTL formula ϕ if and only if the trace t satisfies the formula ϕ at the initial step $k = 0$.

The only difference of the finite semantics from the standard infinite semantics is the definition of the Until ($\mathcal{U}$) operator. The finite $\mathcal{U}$ operator must witness its final condition ϕ' before the test ends. We give additional definitions in Table 3.3(b). These additional definitions are useful in monitoring LTL specifications through a technique called progression. Progression techniques update the LTL formula after every step, in other words, makes progress until the formula simplifies to $\top$ or $\neg\top$.

An example LTL specification is

$$\phi = \bigcirc\Diamond(p \wedge \bigcirc\Diamond q), \tag{3.3}$$

where p and q are Boolean propositions. A test t witnesses this specification if and only if before the test ends, first p, then q becomes true ($\top$). Note that the existence of an order between p and q helps us defining the consecutive tasks of a test scenario, and then monitor those tasks.

4. EXPLORING ANDROID APPLICATIONS WITH QBE

Even with the ongoing development in the state-of-the-art, a simple black-box random testing tool, Monkey [25], proved to perform better than complex tools in terms of coverage and the number of crashes detected [14,17]. The downside of Monkey is that tests it generates are hard to reproduce and faults are hard to localize. Furthermore, Monkey may cause a crash because it's generating an unrealistic action sequence that is not reproducible by hand.

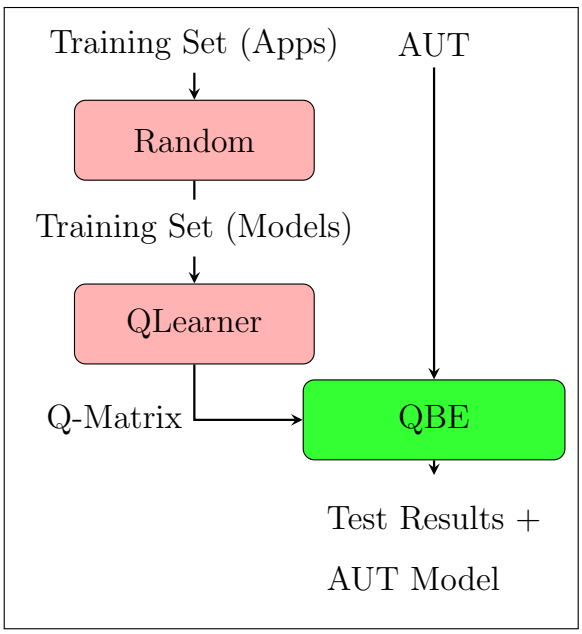


Figure 4.1. QLearning-Based Exploration (QBE) Overview.

We propose QLearning-Based Exploration (QBE) [19], a test generator that explores GUI actions using QLearning [67]. Figure 4.1 shows the overview of QBE. First, QBE crawls through every application in a training set of applications (Apps) by installing, running, and executing random actions on them (Random). During execution, QBE uses a variant of L^* algorithm called PassiveLearn [15] to infer a finite transition system (Model) that approximately fits the AUT. After generating a model for every training application, QLearner samples transitions from these models. Every sampled transition yields a reward value that is either zero and one, meaning the transition is not achieving any objective and accomplishes an objective, respectively. QBE trains

a Reinforcement Learning (RL) agent using these rewards. Note that the Application Under Test (AUT) is not included in the training set. Hence, we say that QBE performs *offline* learning.

QBE learns an action generation policy in the form of a Q-Matrix from all the finite transition models. When Q-Matrix is ready, QBE switches to full exploitation mode, stops learning, and generates tests for the AUT using this policy. Finally, QBE outputs coverage and the number of unique crashes detected (Test Results) along with the inferred transition system of the AUT (AUT Model). Later, we can include the AUT in the training set to improve QBE's performance for newer applications.

QBE needs an automated test generator and executor to crawl the training set and then generate tests for the AUT. For this purpose, we create a modular Android testing and automation framework called AndroFrame. AndroFrame implements not just QBE, but Random Exploration (RE) and Depth-First Exploration (DFE) strategies, as well. Furthermore, AndroFrame reproduces any test it generates, enabling us to verify test results. AndroFrame collects both coverage and unique crash information for evaluation purposes.

4.1. Method

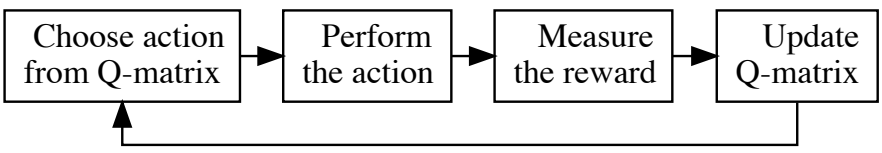


Figure 4.2. Main Flow of QLearner.

Figure 4.2 shows the main flow of QLearner. QLearner keeps a table of values called *Q-matrix*, which is initially all zeros. Every row and column of a Q-matrix denote a state and action, respectively. Hence, every value in a Q-matrix represents the expected future reward of picking the given action at the given state.

The total number of distinct states and actions in an Application Under Test (AUT) is huge, leading to a very large Q-matrix. A large Q-matrix is not just hard to

store but also takes longer to approximate the true values for expected future rewards. For this reason, we abstract out states and actions to a few categories to keep the Q-matrix small. Throughout this section, we denote a state and an abstract state as v and s, respectively. Similarly, we denote an action and an abstract action as z and a.

At every state, QLearner first chooses one abstract action category a with a probability propotional to that abstract action's Q-value. Initially, every Q-value is zero, meaning that every abstract action has the same chance of being chosen. Then QLearner randomly picks an action z whose abstract action is a. Then, QLearner performs z on the device and measures the immediate reward according to an objective function $o(v, z)$, which gives 1 if the objective is reached or 0, otherwise. This objective is either discovering a unique crash or an activity. Finally, QLearner updates its Q-matrix as

$$\underbrace{\vec{Q}[s,a]}_{\substack{\text{Next}\\ \text{Q-Value}}} \leftarrow \underbrace{\vec{Q}[s,a]}_{\substack{\text{Previous}\\ \text{Q-Value}}} + \underbrace{\vec{N}[s,a]^{-1}}_{\substack{\text{History}\\ \text{Value}}} \left(\underbrace{o(v,z)}_{\substack{\text{Objective}\\ \text{Function}}} + \underbrace{\gamma \vec{Q}[s',a']}_{\substack{\text{Future}\\ \text{Expectancy}}} - \underbrace{\vec{Q}[s,a]}_{\substack{\text{Previous}\\ \text{Q-Value}}} \right) \tag{4.1}$$

To calculate updates, QLearner keeps a history matrix called the N-matrix. Every value in the N-matrix is the count of occurences of the given abstract state and action over the number of steps taken so far. Furthermore, QLearner choses a future abstract state and action (s', a'). Finally, it keeps a discount factor λ to decrease the effect of future rewards with the distance of that reward. After enough updates, the Q-matrix converges to the true expected future rewards, which is the optimal action selection policy.

4.2. Evaluation

In this section, we evaluate our new exploration strategy QBE when trained for increasing activity coverage (QBEa) and when trained for detecting crashes (QBEc) by answering two research questions:

RQ1: Activity Coverage. What is the performance of QBEa compared to other black-box Android testing tools in terms of activity coverage?

RQ2: Crash Detection. What is the performance of QBEc compared to other black-box Android testing tools in terms of detection of distinct crashes?

We perform experiments on 300 AUTs we randomly selected from F-Droid benchmark suite [12]. We train QBE algorithm on 200 AUTs for both crash detection and increasing activity coverage. Using the remaining 100 AUTs as test set, we compare QBE with Monkey, PUMA, SwiftHand, Sapienz, DynoDroid, and Depth-First Exploration (DFE) and Random Exploration (RE) strategies of AndroFrame. Note that we never use any applications from the test set in the training set.

4.2.1. Experimental Environment

Table 4.1. Emulators for Testing Tools.

Tool	**Emulator Image**
AndroFrame	Android 4.4.r5 x86 VirtualBox Guest
A^3E	Android 4.4.r5 x86 VirtualBox Guest
Monkey	Android 4.4.r5 x86 VirtualBox Guest
PUMA	Android 4.4.r5 x86 VirtualBox Guest
SwiftHand	Android 4.4.r5 x86 VirtualBox Guest
Dynodroid	ARM (v2.3.3, API 10)
Sapienz	Intel (v4.4.2, API 19)

We performed experiments on an Intel x86 machine with 1TB harddisk, 8x1.6 GHz CPUs containing 8MB L3 cache and running Ubuntu 12.04 operating system. We installed A^3E, Dynodroid, PUMA, SwiftHand, and Sapienz as the state-of-the-art for Android testing tools. We use Android SDK version 25.2.4 and an Android 4.4.r5 x86 image on VirtualBox, since this configuration is compatible with most of the testing

tools. The publicly available versions of Sapienz and Dynodroid are designed to work with the standard Android Emulator [68] and not the VirtualBox image. Hence, we used the Android Emulator to execute Sapienz and Dynodroid. Table 4.1 summarize our experimental environment.

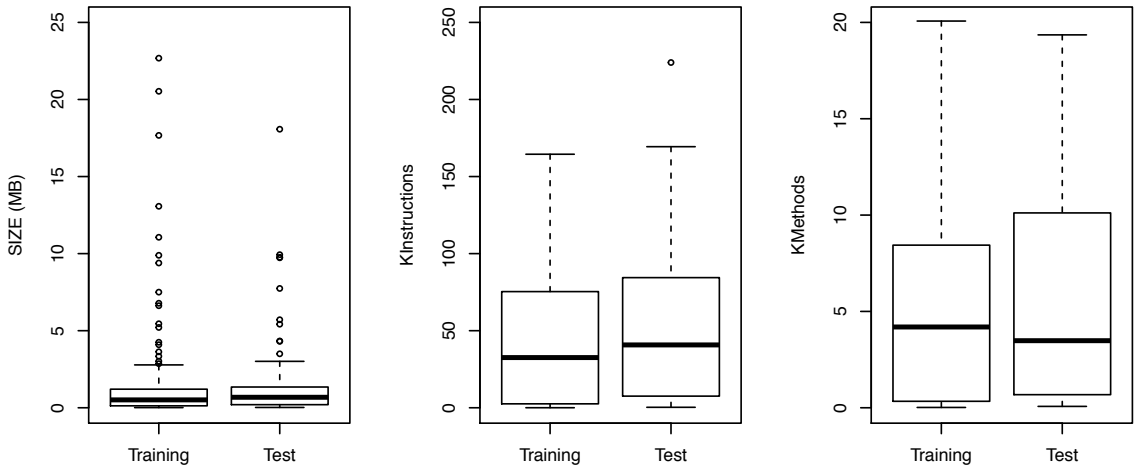


Figure 4.3. AUT Characteristics of Training Set, Test Set and F-Droid Benchmarks.

We downloaded a total of 300 random AUTs from F-Droid benchmark suite [12]. We formed a training set of 200 AUTs out of 300 with random selection. Then, we formed our test set using the remaining 100 AUTs. We compare the characteristics of our training and test sets in Figure 4.3. Box plots show that both the training set and the test set have similar characteristics in terms of application size (in megabytes), number of instructions (in thousands), and number of methods (in thousands). Finally, Table A.1 shows more details on the AUT characteristics of QBE experiments.

4.2.2. State and Action Abstraction

The strength of QBE comes from its ability to learn common patterns across different Android applications to generate tests for more coverage or more crashes. We abstract out GUI states and actions to facilitate learning these common patterns. We divide the state space into 5 abstract states according to the number of enabled actions in the state using the functions on left side of Equation (4.2). We propose these

abstract states by inspecting the mean and variance of the states that we encounter while executing Random Exploration (RE). Similarly, we divide actions into 7 abstract actions as

$$\beta(v) = \begin{cases} 1, & |\lambda(v)| \leq 1 \\ 2, & |\lambda(v)| \leq 3 \\ 3, & |\lambda(v)| \leq 8 \\ 4, & |\lambda(v)| \leq 15 \\ 5, & |\lambda(v)| > 15 \end{cases} \quad \alpha(z) = \begin{cases} 1, & z \text{ is a } \textit{menu} \\ 2, & z \text{ is a } \textit{back} \\ 3, & z \text{ is a } \textit{click} \\ 4, & z \text{ is a } \textit{longclick} \\ 5, & z \text{ is a } \textit{text} \\ 6, & z \text{ is a } \textit{swipe} \\ 7, & z \text{ is a } \textit{contextual} \end{cases} \tag{4.2}$$

4.2.3. Experimental Results

Table 4.2. Experimental Results to answer RQ1 and RQ2.

Tool	RQ1: Coverage Activity	RQ2: Crash # Crashes
DFE	63	3
RE	58	3.2
QBEa	**78**	7.8
QBEc	65	**12.6**
A^3E	41	8
DynoDroid	50	5.2
Monkey	60	9
PUMA	64	6
Sapienz	76	4
SwiftHand	40	0

We execute all testing tools, each for 10 minutes for every AUT in the test set. We also repeat every execution five times, to eliminate the effect of randomness on our experimental results. Table 4.2 shows the resulting coverages and the number of

unique crashes found, averaged over five executions.

Table 4.2 shows that QBEc does not achieve as much coverage as QBEa. Still, QBEa and QBEc achieve the best results in their respective objectives, activity coverage and unique crash detection.

Overall, QBE achieves the highest activity coverage, given that the QLearning algorithm is trained for increasing activity coverage. Also, QBE detects the largest number of unique crashes, given that the QLearning algorithm is trained for crash detection.

4.2.4. Examples from QBE Studies

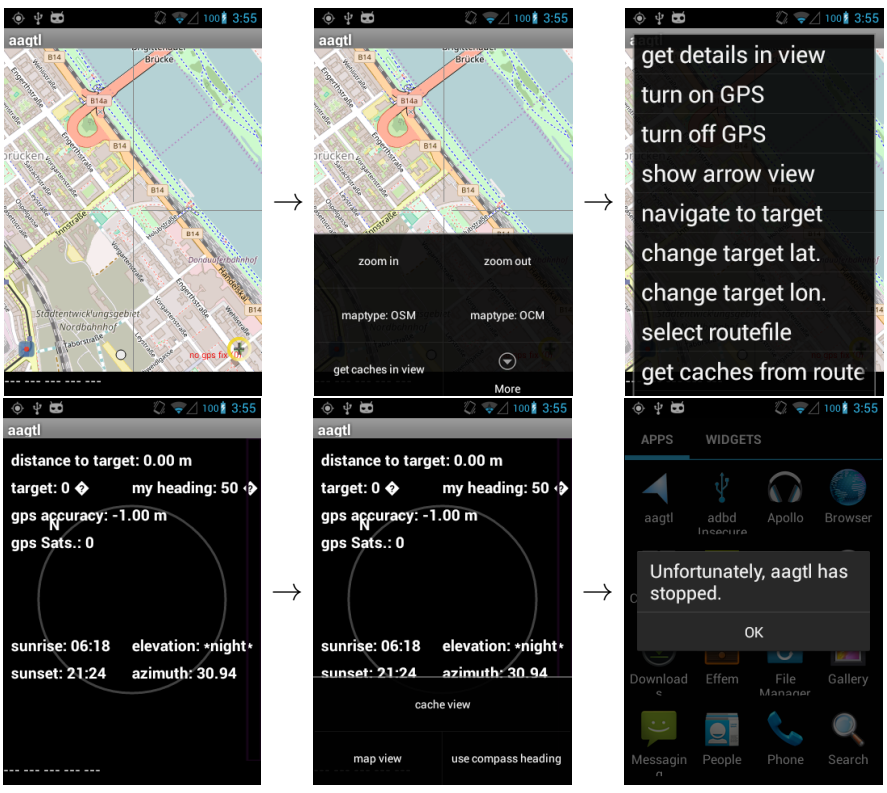


Figure 4.4. A Crashing Test Case of *aagtl* Application.

We investigate a crashing test case for *aagtl* application generated by QBEc within 10 minutes in Figure 4.4. First, we execute a *menu* action to open up the bottom pop-up menu. Then, we click on *More* button to the bottom-right of the screen. Then, we click on *show arrow view* from the list. Now, a black screen with a circle on it

appears. From here, we again execute a *menu* action and click on *cache view* button. Only after these operations, *aagtl* crashes. None of the other tools (PUMA, SwiftHand, Sapienz, Monkey, DynoDroid), including our other methodologies (QBEa, RE, DFE) could detect this crash in 10 minutes. QBEc has a higher chance of finding this crash because it gives higher priority to the *menu* action when there are lots of enabled actions on a screen.

We describe an example case where QBEa reaches more activities. In this case, the AUT is *cz.hejl.chesswalk*, a chess engine application. This application has 10 activities, where QBEa reaches 9 and Sapienz reaches 8 activities. The ninth activity QBE explores is a settings activity. Other testing tools fail to find this activity. Exploration methodologies implemented in RE, PUMA, Sapienz, Dynodroid, and Monkey can not reach the settings activity in the given time, because reachability of the settings activity requires a specific sequence of transitions to be executed, which is hard to hit by random exploration or genetic algorithms. Systematic exploration techniques implemented in DFE and SwiftHand also fail to reach these activities, since they have to exhaust many other sequences before reaching these activities. Similarly, while QBEa is focused on getting to a specific settings activity, Sapienz plays the game more. Hence, since most of the instructions concentrate on the game activity, Sapienz achieves higher instruction coverage. This example leads us to believe that the skewness of distribution of instructions over activities of an application is a possible important characteristic for directing test execution.

4.3. Notes on QBE

QBE's ELTS generator (PassiveLearn) is almost the same as SwiftHand's [15] ELTS generator. The main difference is that QBE's PassiveLearn heavily avoids creating new states and attempts to connect existing states as much as possible. QBE's PassiveLearn achieves this by comparing states using cosine similarity as in PUMA [13].

RE coverage results are similar to Monkey results. This shows us that our testing

framework has no significant flaws compared to Monkey, which also performs random exploration. A^3E and SwiftHand have the worst coverage. We believe one reason for this is because the number of actions supported by A^3E and SwiftHand is small compared to other tools.

The closest competitors of QBE in terms of coverage and crash detection are Sapienz and Monkey, respectively. Statistical tests [19] on the experimental results strengthens the argument that QBE significantly outperformed Sapienz and Monkey.

5. IMPROVING CRASH DETECTION WITH TCM

The main idea of Test Case Mutation (TCM) [21] is to mutate existing test cases to produce richer test cases in order to increase the number of unique crashes detected. We start by investigating typical crash patterns for Android GUI applications. Then, we propose six mutation operators based on these crash patterns. These mutation operators modify actions of a test, which is different than a typical mutation operator that modifies the source code. TCM's mutations convert well-behaving tests into bad-behaving ones, increasing the probability of uncovering a crash.

Unhandled Exceptions, External Errors, Resource Unavailability, Semantic Errors, and Network-Based Crashes are well-known crash patterns for Android GUI applications [20]. TCM's mutation operators are all related to these crash patterns

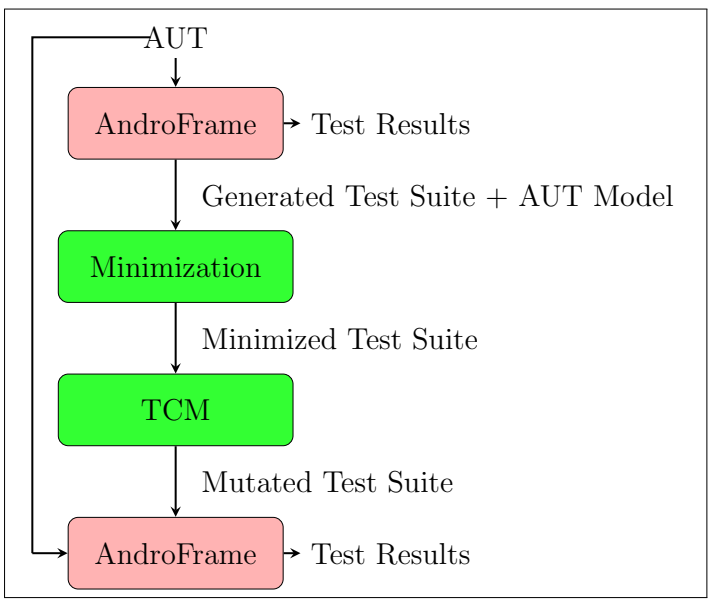


Figure 5.1. Test Case Mutation (TCM) Overview.

Figure 5.1 presents the overview of TCM. First, TCM generates a test suite for the Application Under Test (AUT) using AndroFrame. Note that AndroFrame itself is just a framework, for example, it uses QBE to generate this test suite. Then, AndroFrame obtains an AUT Model as an Extended Labeled Transition System (ELTS). Then, we eliminate tests that do not fit TCM's coverage criteria (Minimization). We perform

this minimization to optimize the execution costs and enable more mutations within a fixed testing time. We apply Test Mutation (TCM) on the Minimized Test Suite and obtain a Mutated Test Suite. Finally, AndroFrame executes the Mutated Test Suite and collects Test Results in terms of the number of unique crashes detected.

5.1. Android Crash Patterns and Mutation Operators

In this section, we first explain known crash patterns for Android applications [20]. Then, we propose six mutation operators related to these patterns.

5.1.1. Android Crash Patterns

5.1.1.1. C1: Unhandled Exceptions. An AUT may crash due to misuse of libraries or GUI components, e.g. overuse of a third party library (stressing) may cause the third party library to crash.

5.1.1.2. C2: External Errors. This error occurs while the AUT is communicating with other applications using Inter Process Communication (IPC), if the AUT

(i) does not have the necessary permissions,
(ii) recieves an invalid message,
(iii) sends an invalid message and crashes other applications, or
(iv) accesses a shared memory segment another application freed.

5.1.1.3. C3: Resource Unavailability. An Android AUT may pause and send itself to background at anytime. The *onPause()* event depends on the AUT's implementation. So, a buggy AUT may crash when paused and resumed, especially if done repeatedly.

5.1.1.4. C4: Semantic Errors. An AUT may crash if it recieves unexpected user input. For example, AUT may crash due to a necessary text field being empty, if the input is

not handled correctly.

5.1.1.5. C5: Network-Based Crashes. An AUT may make remote connections via *bluetooth* or *wifi*. The AUT may crash if it does not handle unreachable servers or disabled WiFi.

5.1.2. Mutation Operators

We now present the six mutation operators for TCM.

5.1.2.1. M1: Loop-Stressing. Loop-stressing repeats all looping actions of a test t multiple times with a mutated delay. Note that shortening the delay of an action may change its nature. For example, a repeated single-click with no delay becomes a double-click.

Our case studies show that stressing an action nine or more times may lead to a crash. So, this operator repeats the mutated action nine times.

Loop-stressing may lead to an unhandled exception (C1) due to stressing the third party libraries by invoking them repeatedly. Loop-stressing may also lead to an external error (C2) if it stresses another application until it crashes.

5.1.2.2. M2: Pause-Resume. This mutation inserts *doze off*-*doze on* actions between every delated transition of a test, with two seconds delay for both inserted actions. Pause-resume may trigger a crash due to resource unavailability (C3).

5.1.2.3. M3: Change Text. We assume the text inputs of original tests are well-behaving. The main idea of this mutation operator is to replace those text inputs with unexpected ones.

This mutation operator randomly

(i) Empties a textbox instead of writing something to it,
(ii) Tries to put a dot character (.) instead of the expected input, or
(iii) Tries to put an extraordinarily long text (longer than 200 characters).

This operator may crash an AUT because the corresponding *onTextChange()* method of the AUT throws an unhandled exception (C1). The AUT may also crash if the content of the text is an unexpected kind of input, which causes a semantic error later (C3).

5.1.2.4. M4: Toggle Contextual State. Our experience shows that some well-behaving tests unexpectedly crash an AUT under different contextual attributes. Hence, this mutation randomly toggles a contextual attribute (GPS, bluetooth, WiFi, etc.) between every transition. Toggling contextual attributes of a state requires system events that take a long time to execute, so we set the delay for every toggle to 10 seconds.

Toggling the contextual states of the AUT may result in an external error (C2), or a network-based crash if the connection failures are not handled correctly (C5).

5.1.2.5. M5: Remove Delays. This mutation operator removes delays from every transition. If the AUT is communicating with another application, removing delays may cause the requests to crash the other application. If this case is not handled in the AUT, the AUT crashes due to external errors (C2). If the AUT's background process is affected by the GUI actions, removing delays may cause the background process to crash due to resource unavailability (C3). If the GUI actions trigger network requests, having no delays may cause a network-based crash (C5).

5.1.2.6. M6: Faster Swipe. This mutation modifies the delays of only Scroll actions. If the information presented by the AUT is downloaded from a network or another

application, swiping too fast may cause a network-based crash (C3) due to the network being unable to provide the necessary data or an external error (C2). If the AUT is a game, swiping too fast may cause the AUT to throw an unhandled exception (C1).

Overall, Table 5.1 shows the relation between crash patterns and TCM mutation operators. Note that some mutations may trigger the same crash pattern.

Table 5.1. Relating Crash Patterns and Mutation Operators.

Crash Patterns	**Mutation Operators**
C1. Unhandled Exceptions	M1, M3, M6
C2. External Errors	M1, M4, M5, M6
C3. Resource Unavailability	M2, M5
C4. Semantic Errors	M3
C5. Network-Based Crashes	M4, M5, M6

5.2. Test Suite Minimization and Test Mutation

Minimizing a test suite has two stages; (i) eliminating unnecessary tests and (ii) trimming the unnecessary actions from the remaining tests. Starting from an empty minimized test suite, we consider every test t of the original test suite. We add t to the minimized test suite only if it improves the overall edge coverage of the minimized test suite. Otherwise, we ignore t. So, the only thing we need for minimization is the edge coverage information. We know that AndroFrame infers an approximate finite state transition system of the AUT. We calculate the edge coverage over this approximated model.

Second, after we pick a test, we eliminate its last action while it does not hinder edge coverage. We trim the test t from its end because otherwise the test t may become invalid with respect to the AUT. We do not wish to make the test bad-behaving at this point, that will be the responsibility of the mutation operators, M1-M6. After the first and second steps, we get a minimized test suite, ready to be mutated.

Require:

TS : A Test Suite

X : Timeout of the New Test Suite

Δ: Mutation Operators

Ensure:

TS' : New Test Suite

1: $TS' \leftarrow \{\}$

2: $x \leftarrow 0$

3: **repeat**

4: $t \leftarrow$ random $t \in TS$ ▷ Pick a random test case

5: $\delta \leftarrow$ random $\delta \in \Delta$ **s.t.** $t \neq \delta(t)$ ▷ Pick a random effective mutation operator

6: $t' \leftarrow \delta(t)$ ▷ Apply the mutation operator on the test case

7: $TS' \leftarrow TS' \cup \{t'\}$ ▷ Add the mutated test case to the New Test Suite

8: $x \leftarrow x + \sum_{(v_s, v_e, z, d) \in t'} d$ ▷ Calculate the total delay

9: **until** $x > X$ ▷ Repeat until the total delay is above the given timeout

Figure 5.2. Test Mutation (TCM) Algorithm.

Figure 5.2 shows our Test Mutation approach. We start with an initially empty mutated test suite TS' and total delay (x) of zero. Then, we consider a random test t in the minimized test suite TS in Line 4. Then, we pick a random effective mutation operator δ from the set of all mutation operators $\Delta = \{M1, M2, M3, M4, M5, M6\}$. A mutation operator δ is effective only if it changes the underlying test, i.e. $t' \neq \delta(t)$. We mutate t with δ and add the mutated test case t' to TS' until the total delay of TS' exceeds the given timeout X. After the timeout X is reached, we terminate because we assume the time given for testing is constant.

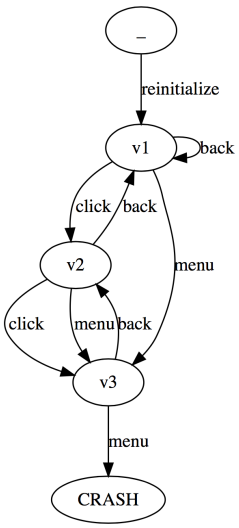
	Test Case A			
1	_	v1	reinit	10
2	v1	v2	click	1
3	v2	v1	back	1
4	v1	v2	click	1
5	v2	v1	back	1

	Test Case B			
1	_	v1	reinit	8
2	v1	v3	menu	2
3	v3	CRASH	menu	1

	Test Case C			
1	_	v1	reinit	9
2	v1	v1	back	0
3	v1	v2	click	1
4	v2	v3	click	2
5	v3	CRASH	menu	2

	Test Case D			
1	_	v1	reinit	15
2	v1	v1	back	0
3	v1	v2	click	2
4	v2	v1	back	1
5	v1	v3	menu	3

(a) Test Cases.



(b) AUT Model.

	Mutated 1			
1	_	v1	reinit	15
2	**v1**	**v1**	**back**	**1**
3	**v1**	**v1**	**back**	**1**
4	**v1**	**v1**	**back**	**1**
5	**v1**	**v1**	**back**	**1**
6	**v1**	**v1**	**back**	**1**
7	**v1**	**v1**	**back**	**1**
8	**v1**	**v1**	**back**	**1**
9	**v1**	**v1**	**back**	**1**
10	**v1**	**v1**	**back**	**1**
11	v1	v1	back	0
12	v1	v2	click	2
13	v2	v1	back	1
14	v1	v3	menu	3

	Mutated 2			
1	_	v1	reinit	15
2	**v1**	_	**doze off**	**2**
3	_	**v1**	**doze on**	**2**
4	v1	v1	back	0
5	**v1**	_	**doze off**	**2**
6	_	**v1**	**doze on**	**2**
7	v1	v2	click	2
8	**v2**	_	**doze off**	**2**
9	_	**v2**	**doze on**	**2**
10	v2	v1	back	1
11	**v1**	_	**doze off**	**2**
12	_	**v1**	**doze on**	**2**
13	v1	v3	menu	3

(c) Mutated Test Cases.

Figure 5.3. Motivating Example (mutations are bold).

5.3. Motivating Example

Figures 5.3(a) and 5.3(b) show a test suite and an AUT model, respectively. We generate this test suite and the AUT model by executing AndroFrame for one minute on an example AUT. We limit the maximum number of transitions per test case to five to keep the test cases small in this motivating example. The test suite has four test cases; A, B, C, and D. Each row of every test case describes a delayed transition. We disregard action parameters for the sake of simplicity.

Among the four test cases reported by AndroFrame, we take only the non-crashing test cases, A and D. In our example, we include D since it increases the edge coverage and we exclude A since all of A's transitions are also D's transitions, i.e. A is subsumed by D. Then, we attempt to minimize test case D without reducing the edge coverage. In our example, we don't remove any transitions from D because all transitions in D contribute to the edge coverage. We then generate mutated test cases by randomly

applying mutation operators to D one by one until we reach one minute timeout. Figure 5.3(c) shows an example mutated test suite. Test case Mutated 1 takes D and exercises the back button for multiple times to stress the loop at state v1. Test case Mutated 2 clicks the hardware power button twice (doze off, doze on) between each transition. This operation pauses and resumes the AUT in our test devices. We then execute all mutated test cases on the AUT. Our example AUT in fact crashes when the loop on v_1 is reexecuted more than eight times and also crashes when the AUT is paused in state v_2. When executed, our mutated test cases reveal these crashes both at their ninth transition, doubling the number of detected crashes.

5.4. Evaluation

In this section, we evaluate TCM through experiments and case studies, showing that how we detect crash patterns and improve crash detection.

5.4.1. Experiments

We used the QBE's experimental test set of 100 AUTs. This time, we execute every test generator 20 minutes instead of 10 because we aimed to reuse the AUT models QBEc (QBE for crash) generated for the QBE experiments. Hence, for TCM, we executed QBEc for only 10 minutes, and then switched to TCM.

Figure 5.4 shows the change in the number of total unique crashes detected by TCM (10 mins QBEc + 10 mins TCM), ANDROFRAME (20 mins QBEc), SAPIENZ, MONKEY, PUMA, and A^{3}E. Our results confirm that pure QBEc detects more crashes than any other tool from very early on. TCM detects the same number of crashes with AndroFrame for the first 10 minutes (600 seconds) because we use QBEc for its first 10 minutes. At the point where we switch from QBEc to TCM (10 minute mark), QBEc already have found 15 unique crashes. During the remaining 10 minutes, TCM detects 14 more crashes whereas QBEc detects only 3 more crashes. As a result, TCM detects 29 crashes in total whereas QBEc detects 18 crashes in total. As a last note,

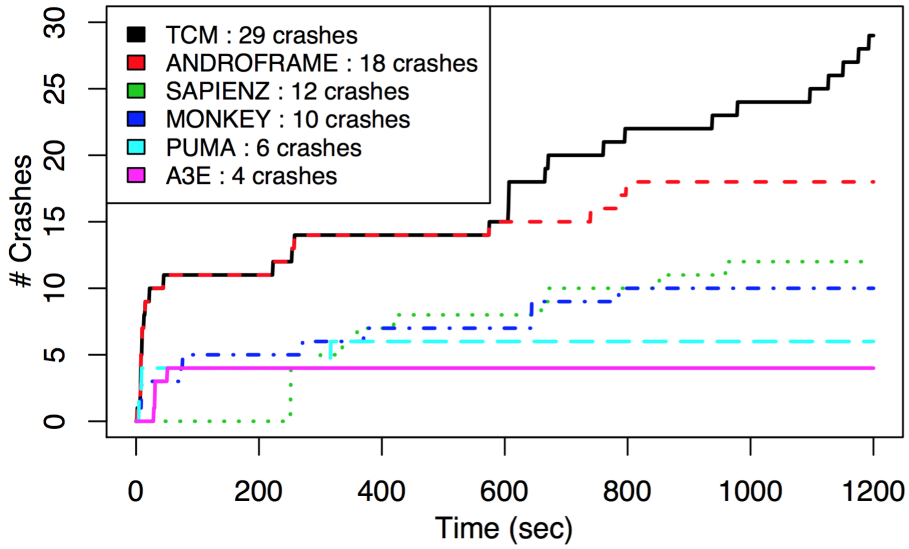
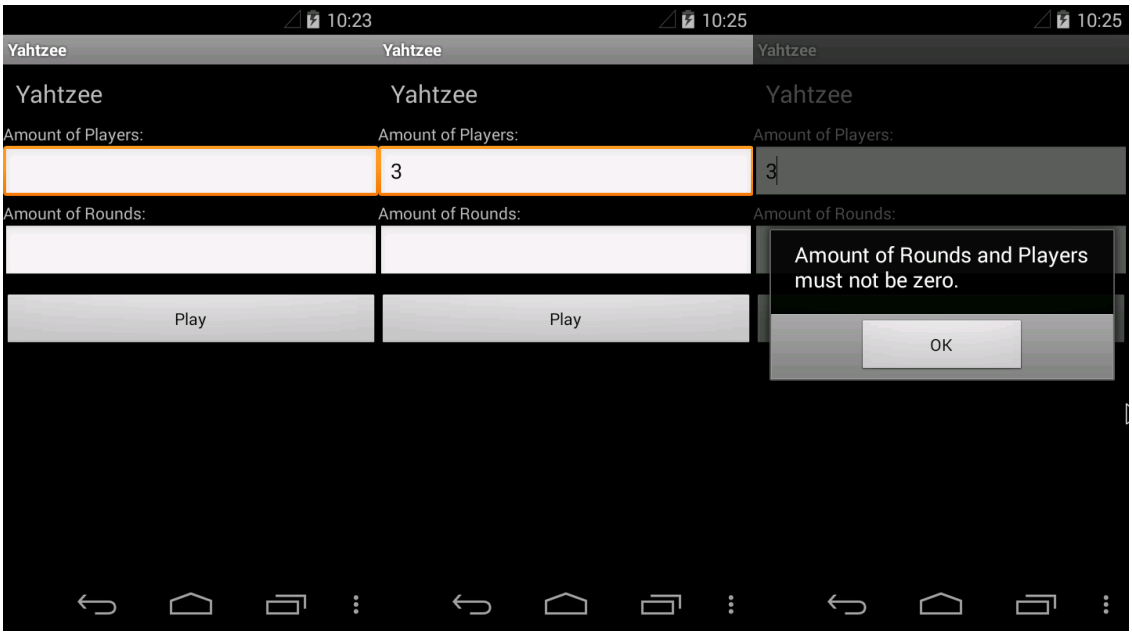


Figure 5.4. Number of Total Distinct Crashes Detected Across Time.

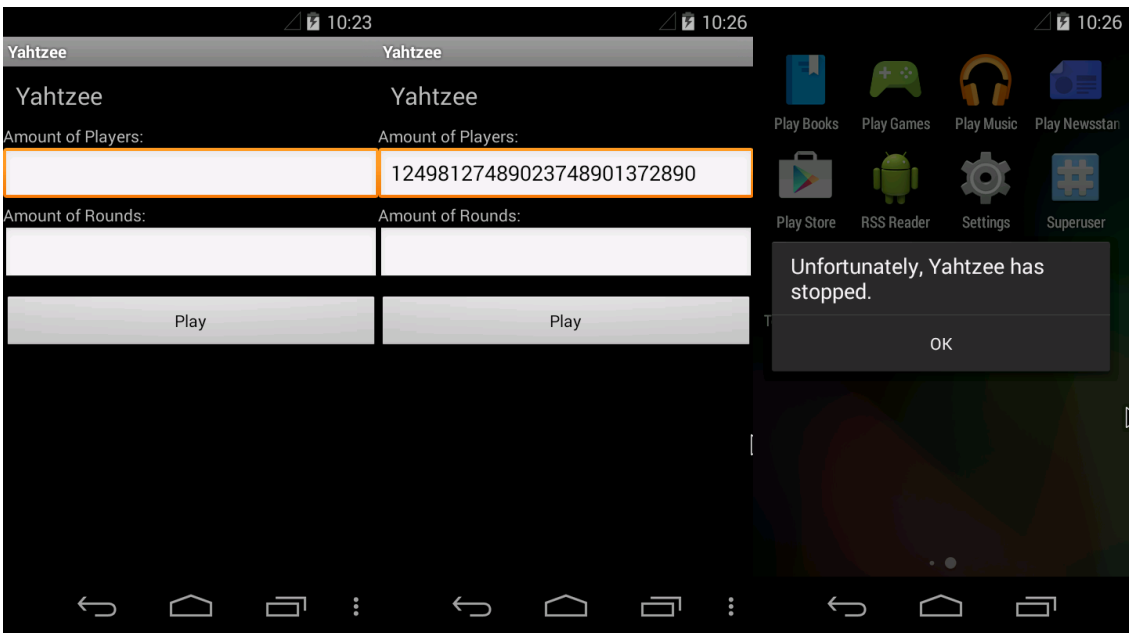
all other tools including QBEc seem to stabilize after 20 minutes whereas TCM finds many crashes near timeout. This shows us that TCM may find even more crashes when timeout is longer. Overall, TCM improves unique crash detection even further than QBEc.

We also investigate how much each mutation operator contributes to the number of detected crashes. Our observations reveal that M1 detects one crash, M2 detects four crashes, M3 detects two crashes, M4 detects two crashes, M5 detects four crashes, and M6 detects one crash. These crashes add up to 14, which is the number of crashes detected by TCM in the last 10 minutes. This result shows that while all mutation operators contribute to the crash detection, M2 and M5 have the largest contribution.

We present and explain one crash that is found only by TCM in Figure 5.5. Figure 5.5(a) shows an instance where AndroFrame generates and executes a test case t on the Yahtzee application. Note that t does not lead to a crash, but only a warning message. Figure 5.5(b) shows the instance where TCM mutates t and executes the mutated test case t'. When t' is executed, the application crashes and terminates. We note that this crash was not found by any other tool. Mao et al. [16] also report that Sapienz and Dynodroid did not find any crashes in this application.



(a) Execution of Test Case t.



(b) Execution of Test Case $t' = M3(t)$.

Figure 5.5. An Example Crash Found Only by TCM.

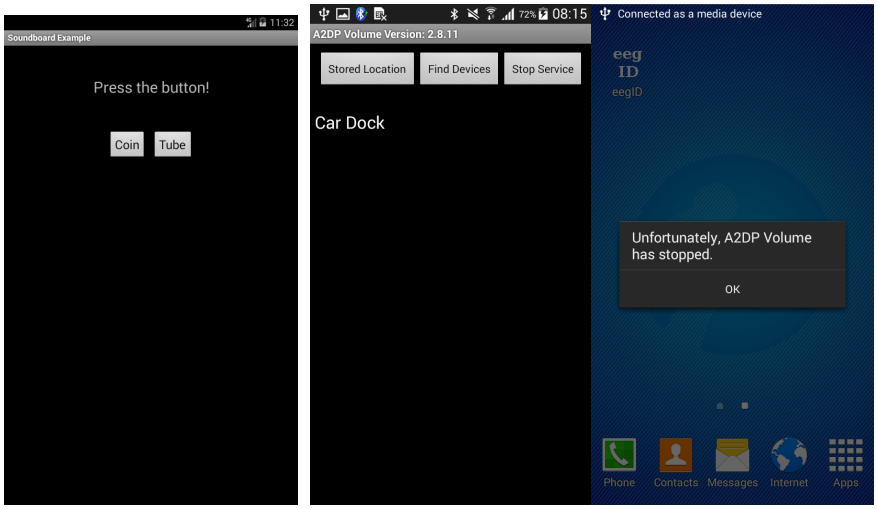
5.4.2. Case Studies

In this section, we verify that the crash patterns we use in TCM exist via case studies, one case study for each crash pattern. These studies verify that all of our crash patterns are observable in Android, facilitating the development and fine-tuning of our mutation operators.

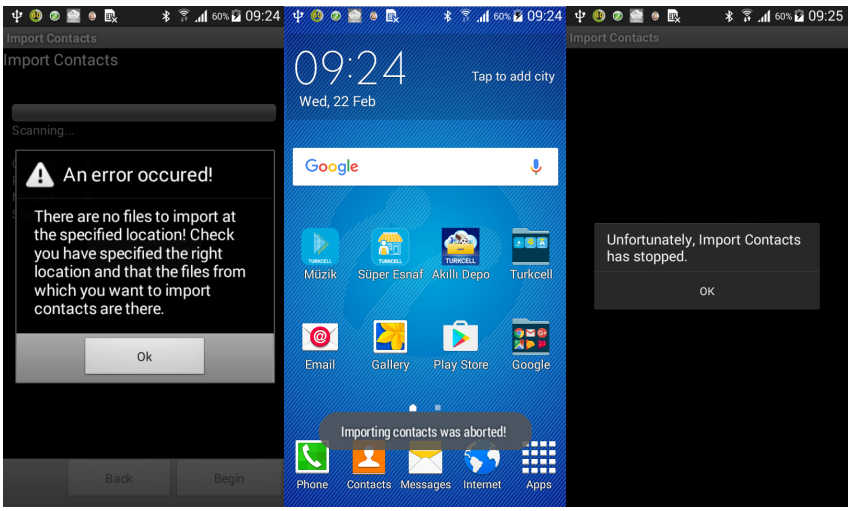
5.4.2.1. Case Study 1. Figure 5.6(a) shows a crashing activity of the *SoundBoard* application included in F-Droid benchmarks. Basically, the *coin* and *tube* buttons activate a third party library, AudioFlinger, to produce sound when tapped. AndroFrame generates test cases which tap these buttons. These test cases produce no crashes. Then, we mutate the test cases with TCM. When we apply *loop-stressing* (M1) on any of these buttons, AudioFlinger crashes due to overuse. AudioFlinger produces a fatal exception (C1) in Android logs. This crash does not cause an abnormal termination, but it causes the AUT to stop functioning (the AUT stops producing sounds until it is restarted).

5.4.2.2. Case Study 2. Figure 5.6(b) shows a crashing activity of the *a2dpVol* application included in F-Droid benchmarks, where AndroFrame fails to generate crashing test cases. We mutate these test cases with TCM. When we activate bluetooth (M4), tapping *find devices* button produces a crash in the external *android.bluetooth.IBluetooth* application due to a missing method (C2) and the AUT terminates.

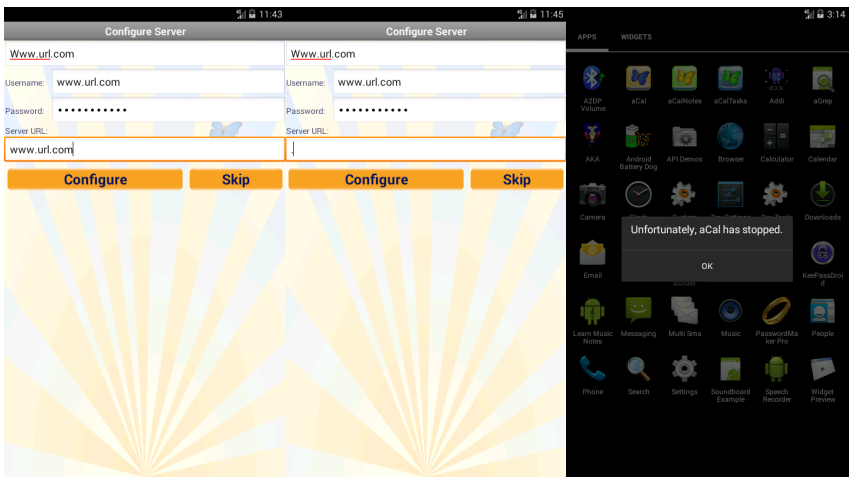
5.4.2.3. Case Study 3. Figure 5.6(c) shows a crashing activity of the *importcontacts* application included in F-Droid benchmarks. The AUT handles the case that it fails to import contacts, as we show in the leftmost screen. Pausing the AUT at this screen causes the background process to abort and free its allocated memory (we show the related screen in the middle). However, the paused activity is not destroyed. If the user tries to resume this activity, the AUT crashes as we show in the rightmost screen, since the memory was freed before. TCM applies a pause-resume mutation (M2) and



(a) Unhandled Ex. (C1). (b) External Error (C2).



(c) Resource Unavailability (C3) Example.



(d) Semantic Error (C4) Example.

Figure 5.6. Case Studies 1-5.

triggers this resource unavailability crash (C3).

5.4.2.4. Case Study 4. Figure 5.6(d) shows a crashing activity of the *aCal* application included in F-Droid benchmarks. AndroFrame generates test cases with well-behaving text inputs. These test cases produce no crashes. Then, we mutate the test cases with TCM. When we apply *change text* (M3) on the last text box and then tap the *configure* button, this produces a semantic error (C4). The AUT crashes and terminates.

5.4.2.5. Case Study 5. Finally, we studied a crashing activity of the *Mirrored* application included in F-Droid benchmarks. When *wifi* is turned off, the AUT goes into offline mode and does not crash as shown in the leftmost screen. When we toggle *wifi* (M4), the AUT retrieves several articles but crashes when it fails to retrieve article contents due to a network-based crash (C5).

5.5. Notes on TCM

There are two challenges about generalizing TCM to different environments (e.g., iOS, Web Browsers, etc.):

(i) Our crash patterns are Android-specific and may not be available on different environments and
(ii) M1-M6 may not be applicable to those platforms,

Hence, every GUI environment may require its own TCM study to determine patterns and related mutations.

One advantage of TCM is that it can be used in combination with any test generator. However, TCM requires an initially generated test suite, so it cannot be a stand-alone test generator.

We did not encounter any crash patterns other than the five crash patterns that we describe in Section. However, it is still possible to observe other crash patterns with our mutation operators due to emerging crash patterns caused by the fragmentation and fast development of the Android platform.

Although TCM detects crashes, it does not detect all possible bug patterns. Qin et al. [69] thoroughly classifies all bugs in Android. According to this classification, there are two types of bugs in Android, Bohrbugs and Mandelbugs. A Bohrbug is a bug whose reachability and propagation are simple. A Mandelbug is a bug whose reachability and propagation are complicated. Qin et al. further categorize Mandelbugs as Aging Related Bugs (ARBs) and Non-Aging Related Mandelbugs (NAMs). Qin et al. also define five subtypes for NAMs and six subtypes for ARBs. TCM detects only the first two subtypes of NAMs, TIM and SEQ. TIM and SEQ are the only kinds of bugs which are triggered by user inputs. If a bug is TIM, the error is caused by the timing of inputs. If a bug is SEQ, the error is caused by the sequencing of inputs.

Our mutation operators insert multiple transitions to the test case, creating an issue of locating the fault inducing transition. Given that the mutated test case detects a crash, fault localization can be achieved using a variant of *delta debugging* [70].

6. FUNCTIONAL TESTING WITH FARLEAD

A recent survey shows that 78% of mobile GUI application users regularly encounter bugs that cause the GUI application to fail at performing some of its functions [71]. Testing if the GUI application performs its intended functions correctly is essential for mitigating this problem.

We could perform model checking on an AUT Model to prove a functional behavior but model checking is costly and not scalable enough. So, we are left with only one option, to verify the functional behavior through witnessing related a test scenario. Previously, test generators use what we call an implicit test oracle [72] where no input is needed to automate the evaluation of a test execution. Now, the test oracle must monitor a test scenario in run-time. Then, the test scenario is an input to the test oracle. In other words, the test oracle becomes specified instead of implicit and the test scenario is the specification.

Typically, a test scenario is a human-readable, natural language description. However, such test scenarios are bound to be ambiguous, therefore unmonitorable. Hence, a run-time monitorable test scenario must be a formal specification.

Even if we have a formal specification as a test scenario and we can monitor it in run-time, we still need a test generator driven towards witnessing the test scenario. However, functional behaviors, in contrast to common crashes and activity patterns, are AUT specific. Therefore, offline learning is not a solution as was in QBE. Any functional test generator must derive AUT specific heuristics to be effective at producing witnesses.

We propose Fully Automated Reinforcement LEArning Driven (FARLEAD) test generator. FARLEAD is an online RL engine. In other words, it learns a new action generation policy for every AUT, from scratch. FARLEAD infers a reward function

from its test scenario monitor. The test scenario monitor accepts a Linear-time Temporal Logic (LTL) formula as a formal specification. Note that we devise finite semantics for LTL instead of the typical infinite semantics because the tests we deal with are all finite.

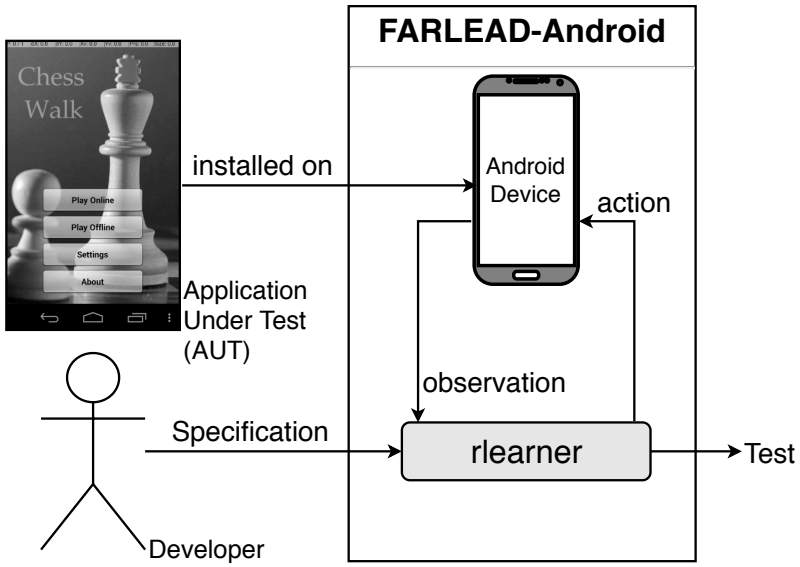


Figure 6.1. FARLEAD Overview.

Figure 6.1 shows the overview of FARLEAD. FARLEAD assumes (i) there is an Android Device with the AUT installed on and (ii) the developer/tester inputs an LTL test scenario. *rlearner* takes the test scenario as input, executes actions on the Android Device, receiving observations from it, and producing reward values by monitoring the test scenario. Once a candidate test satisfies the LTL specification according to its finite semantics, FARLEAD outputs the test as a witness.

Typically, a reward function only rewards when a goal is achieved. This is too late for FARLEAD because once a witness is generated, FARLEAD is no longer needed, the developer/tester already has the witness. Hence, the underlying reward function must reward the progress of the *rlearner*. So, we propose LTL-based Reward Shaping that produces intermediate rewards for a test scenario, driving the execution towards the goal before the goal is reached.

We summarize our contributions with FARLEAD as follows:

(i) To the best of our knowledge, FARLEAD is the first fully automated functional mobile GUI test generator.

(ii) We implement a novel improvements in RL in the form of LTL-based Reward Shaping.

(iii) We evaluate FARLEAD via experiments on two applications from F-Droid. We show that our approach outperforms and is more effective than three known test generation approaches, namely Monkey, Random, and QBEa.

6.1. FARLEAD Methodology

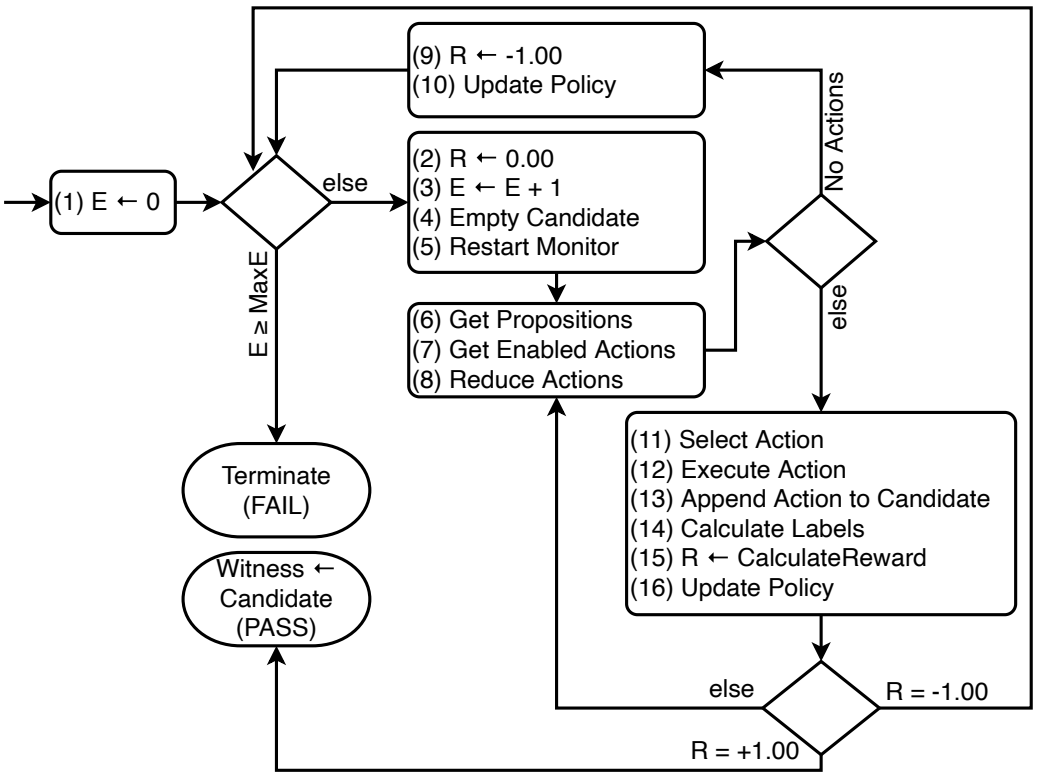


Figure 6.2. FARLEAD Flowchart.

We explain the methodology behind FARLEAD with the flowchart in Figure 6.2. First, Line (1) initializes the number of episodes (E) to zero. At every episode, FARLEAD will produce one candidate test. If the number of episodes is equal to or larger than a predefined maximum number of episodes (MaxE), FARLEAD terminates because it had failed to generate a witness within given limis. Otherwise, FARLEAD begins a new episode between Lines (2)-(5). Line (2) sets the reward value R to zero.

Line (3) increments the number of episodes. Line (4) empties the candidate. Line (5) restarts the monitor, so the monitor starts monitoring the test scenario from the beginning. FARLEAD starts a new test step between (6)-(8). Line (6) gets the monitored propositions. Line (7) gets the set of enabled actions from the AUT. Line (8) reduces the set of enabled actions according to necessary and sufficient propositions. FARLEAD-Android reaches a dead-end only if there are no enabled actions after reduction. In this case, Line (9) sets R = -1.00, and Line (10) updates the action selection Policy according to R. Then, FARLEAD a new episode. Otherwise, Line (11) selects a GUI action according to the Policy. Line (12) executes this GUI action, and Line (13) appends it to the candidate test. Line (14) calculates the labels. Finally, Line (15) calculates R from these labels, and Line (16) updates the action generation Policy. If R = $+1.00$, the candidate is a witness, and FARLEAD terminates. If R = -1.00, it starts a new episode. Otherwise, FARLEAD proceeds to generate a new step by going to Line (6). FARLEAD eventually terminates because its monitor has an internal step counter that produces a negative reward if there are too many steps in the episode.

6.2. Runtime LTL Monitoring via Progression

FARLEAD accepts LTL specifications as high level test scenarios. The advantage of LTL is that it is run-time monitorable via a known technique called progression [50, 56]. However, a typical progression-based monitoring monitors LTL formulae with infinite semantics. In this section, we provide an algorithm for monitoring LTL specifications with finite semantics.

The algorithm in Figure 6.3 shows our monitor for an LTL specification in finite semantics. After every GUI action, this monitor calls calculateImmediateReward, which in turn progresses the LTL specification for the k^{th} step using the Boolean labels L. After one progression, if the specification becomes true ($\top$), we return a positive one, indicating the test generated witnesses the LTL specification, completely. If the specification becomes ($\neg\top$), the test is a bad prefix and it will not witness the specification no matter what the future GUI actions are. So we return a negative one. Otherwise,

1: **procedure** calculateImmediateReward($k \in \mathbb{N}, L \subseteq \mathcal{AP}$)

2: $\quad \phi_{k+1} \leftarrow \text{progress}(\phi_k, L)$

3: $\quad$ **return** $\begin{cases} 1 & \phi_{k+1} = \top \\ -1 & \phi_{k+1} = \neg\top \\ \frac{\lvert N(\phi_{k+1}) - N(\phi_k)\rvert}{N(\phi_{k+1}) + N(\phi_k)} RS & \text{otherwise} \end{cases}$ ▷ N: Atomic proposition count

4: **end procedure**

5: **procedure** progress(ϕ in LTL, $L \subseteq \mathcal{AP}$)

6: $\quad$ **return** advance(restrict(expand(ϕ), L))

7: **end procedure**

8: **procedure** expand(ϕ in LTL)

9: $\quad$ **return** $\begin{cases} \neg\text{expand}(\phi') & \phi = \neg\phi' \\ \text{expand}(\phi') \wedge \text{expand}(\phi'') & \phi = \phi' \wedge \phi'' \\ \text{expand}(\phi'') \vee (\text{expand}(\phi') \wedge \bigcirc\phi) & \phi = \phi' \mathcal{U} \phi'' \\ \phi & \text{otherwise} \end{cases}$

10: **end procedure**

11: **procedure** restrict(ϕ in LTL, $L \subseteq \mathcal{AP}$)

12: $\quad$ **return** $\begin{cases} \top & \phi = \top \textbf{ or } \phi \in L \\ \neg\top & \phi \in \mathcal{AP} \textbf{ but } \phi \notin L \\ \neg\text{restrict}(\phi') & \phi = \neg\phi' \\ \text{restrict}(\phi') \wedge \text{restrict}(\phi'') & \phi = \phi' \wedge \phi'' \\ \phi & \text{otherwise} \end{cases}$

13: **end procedure**

14: **procedure** advance(ϕ in LTL)

15: $\quad$ **return** $\begin{cases} \text{advance}(\phi') & \phi = \neg\phi' \\ \text{advance}(\phi') \wedge \text{advance}(\phi'') & \phi = \phi' \wedge \phi'' \\ \phi' & \phi = \bigcirc\phi' \\ \phi & \text{otherwise} \end{cases}$

16: **end procedure**

Figure 6.3. Progression based LTL Monitoring with Reward Shaping.

we return a value between zero and one, depending on the amount of change in the LTL specification. Using partial reward values is called Reward Shaping [24].

The algorithm in Figure 6.3 modifies a given LTL specification at every k^{th} test step, using the *progress()* procedure. The *progress()* procedure consecutively applies (i) *expand()*, (ii) *restrict()*, and (iii) *advance()* on the LTL specification. The *expand()* procedure expands every $\mathcal{U}$ operator with a $\bigcirc$ operator as

$$\phi' \mathcal{U} \phi'' = \neg(\neg\phi'' \wedge \neg(\phi' \wedge \bigcirc(\phi' \mathcal{U} \phi'')) \tag{6.1}$$

The *restrict()* procedure replaces all the current state variables with $\top$ or $\neg\top$ according to the current state labels L. This procedure may minimize the whole formula to $\top$ or $\top$. Finally, the *advance()* procedure just removes one level of $\bigcirc$ operators from the specification because all the next states become current states at the $(k+1)^{\text{th}}$ step.

6.3. FARLEAD Example

We now demonstrate how FARLEAD works with a small example on a chess game called ChessWalk. The GUI function that we need to witness is the ability to go from MainActivity to AboutActivity and then return to MainActivity. We specify an LTL test scenario for it as $\phi_0 = \bigcirc(p\mathcal{U}(q \wedge \bigcirc(q\mathcal{U}p)))$, where p and q are true if and only if the current activity matches with the word Main and the word About, respectively. According to ϕ_0, the activity of the second state must be MainActivity until it is AboutActivity and then it must be AboutActivity until it is MainActivity. Note that we start from the second state and not the first because we assume the first state is a don't care state from which we reinitialize the AUT.

In our example, FARLEAD finds a witness in three episodes. An episode starts with a *reinit* action and ends with the final action of a candidate test. Table 6.1 shows these episodes with screenshots in Figures 6.4-6.6. Note that the *reinit* action is the only choice at a don't care state, so every episode begins with a_0 = reinit chesswalk MainActivity. At $k = 0$, FARLEAD executes a_0, observing a set of labels (Boolean propositions) L. Then, it computes a ϕ_{k+1} from ϕ_k using LTL pro-

Table 6.1. FARLEAD Example.

$\mathcal{AP} = \{p = [\text{ activity} \sim \text{Main }], q = [\text{ activity} \sim \text{About }]\}$

	EPISODE $i = 1$		$\phi_0 = \bigcirc(p\mathcal{U}(q \wedge \bigcirc(q\mathcal{U}p)))$	
$k = 0$	$a_0 =$ reinit ChessWalk MainActivity	$L = \{p\}$	$\phi_1 = p\mathcal{U}(q \wedge \bigcirc(q\mathcal{U}p))$	$r = 0$
$k = 1$	$a_1 =$ click Chess	$L = \{p\}$	$\phi_2 = p\mathcal{U}(q \wedge \bigcirc(q\mathcal{U}p))$	$r = 0$
$k = 2$	$a_2 =$ back	$L = \{\}$	$\phi_3 = \neg\top$	$r = -1$
	EPISODE $i = 2$		$\phi_0 = \bigcirc(p\mathcal{U}(q \wedge \bigcirc(q\mathcal{U}p)))$	
$k = 0$	$a_0 =$ reinit ChessWalk MainActivity	$L = \{p\}$	$\phi_1 = p\mathcal{U}(q \wedge \bigcirc(q\mathcal{U}p))$	$r = 0$
$k = 1$	$a_1 =$ click About	$L = \{q\}$	$\phi_2 = q\mathcal{U}p$	$r = .33$
$k = 2$	$a_2 =$ click Link	$L = \{\}$	$\phi_3 = \neg\top$	$r = -1$
	EPISODE $i = 3$		$\phi_0 = \bigcirc(p\mathcal{U}(q \wedge \bigcirc(q\mathcal{U}p)))$	
$k = 0$	$a_0 =$ reinit chesswalk MainActivity	$L = \{p\}$	$\phi_1 = p\mathcal{U}(q \wedge \bigcirc(q\mathcal{U}p))$	$r = 0$
$k = 1$	$a_1 =$ click About	$L = \{q\}$	$\phi_2 = q\mathcal{U}p$	$r = .33$
$k = 2$	$a_2 =$ back	$L = \{p\}$	$\phi_3 = \top$	$r = 1$

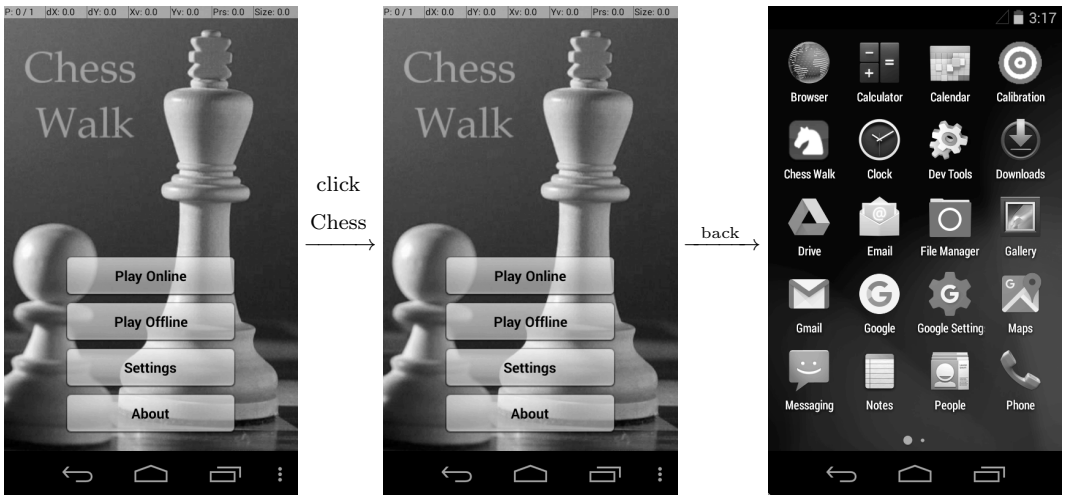


Figure 6.4. Episode i = 1.

Figure 6.5. Episode i = 2.

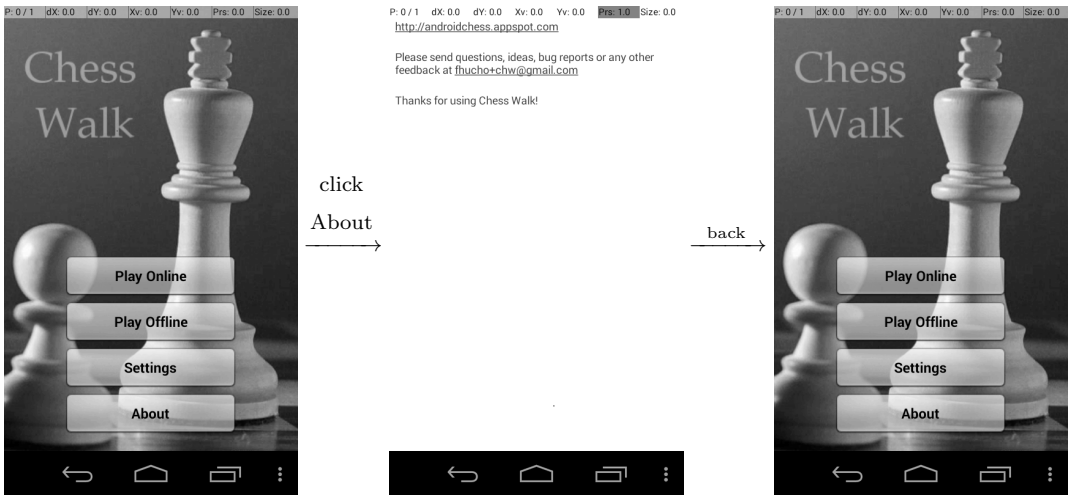


Figure 6.6. Episode i = 3.

gression with these labels L. Since a_0 is always the same, the resulting L is always the same too, where p is true and q is not because the resulting activity matches the word Main. FARLEAD computes the LTL progression on ϕ_0 as in Algorithm 6.3. The resulting formula is $\phi_1 = p\mathcal{U}(q \wedge \bigcirc(q\mathcal{U}p))$, which is different than ϕ_0, which could indicate progress. FARLEAD calculates the amount of progress (immediate reward) as $r = |N(\phi_1) - N(\phi_0)|/(N(\phi_1) + N(\phi_0)) = 0/8 = 0$. The zero reward means that a_0 is neither positie nor negative for witnessing the test scenario.

In the first episode, FARLEAD has no idea which action is going to witness the LTL, so it chooses a random action $a_1 =$ click Chess. This action clicks the Chess text on the AUT, which triggers no events, so we reach the same state, as Figure 6.4 shows. Again, labels are $L = \{p\}$. This time, we expand ϕ_1 as

$$\neg(\neg(q \wedge \bigcirc(q\mathcal{U}p)) \wedge \neg(p \wedge \bigcirc(p\mathcal{U}(q \wedge \bigcirc(q\mathcal{U}p))))) \tag{6.2}$$

$L = \{p\}$ means $p = \top$ and $q = \neg\top$ in the current state. Hence, we restrict ϕ_1 accordingly, as

$$\neg(\neg(\neg\top \wedge \bigcirc(q\mathcal{U}p)) \wedge \neg(\top \wedge \bigcirc(p\mathcal{U}(q \wedge \bigcirc(q\mathcal{U}p))))) \tag{6.3}$$

Note that we do not replace atomic propositions protected by a $\bigcirc$ operator. After we minimize the resulting formula and advance one step, we obtain $\phi_2 = p\mathcal{U}(q \wedge \bigcirc(q\mathcal{U}p)))$. Since $\phi_2 = \phi_1$, we again calculate the immediate reward as zero.

Finally, after executing $a_2 =$ back randomly, we get out of the AUT, as shown in

Figure 6.4, so the current activity is neither MainActivity nor AboutActivity. Therefore, the labeling is empty, so $p = q = \neg\top$. We calculate the final formula as $\phi_3 = \neg\top$. We calculate the immediate reward as $r = -1$ and terminate this episode.

In the second episode, FARLEAD opens AboutActivity, but fails to return to MainActivity, as shown in Figure 6.5. Though it gets $r = -1$ in the end, it also obtains $r = .33$ for opening AboutActivity. The intermediate reward instructs FARLEAD to explore the second action again in the next episode.

In the final episode, FARLEAD again opens AboutActivity and gets $r = .33$ as before. This time, it finds the correct action and returns to MainActivity, as shown in Figure 6.6. Therefore, it receives $r = 1$ and terminates. The action sequence generated in the final episode is the witness.

6.4. Evaluation

In this section, we demonstrate the effectiveness and the performance of FARLEAD through experiments on a VirtualBox guest with Android 4.4 operating system and 480x800 screen resolution. In evaluation, a virtual machine is better than a physical Android device because

(i) anyone can reproduce our experiments without the physical device, and
(ii) even if a physical device is available, it must be the same with the original to produce the same results.

Table 6.2. FARLEAD's Experimental Test Scenarios.

		Source	Description	Level	Avaliable
AUT: ChessWalk	ϕ_A	App-Agnostic	The user must be able to go to AboutActivity and return back.	(a)	✔
				(b)	✔
				(c)	✔
	ϕ_B	App-Agnostic	The user must be able to go to SettingsActivity and return back.	(a)	✔
				(b)	✔
				(c)	✔
	ϕ_C	App-Agnostic	Pausing and resuming the AUT should not change the screen.	(a)	✖
				(b)	✔
				(c)	✔
	ϕ_D	Bug Reports	The AUT should prevent the device from sleeping but it does not.	(a)	✔
				(b)	✔
				(c)	✔
	ϕ_E	Bug Reports	The AUT should prevent the device from sleeping but it does not.	(a)	✔
				(b)	✔
				(c)	✔
	ϕ_F	Manually Created	The AUT should prevent the device from sleeping but it does not.	(a)	✔
				(b)	✔
				(c)	✔
	ϕ_G	Novel Bug	The AUT should prevent the device from sleeping but it does not.	(a)	✔
				(b)	✔
				(c)	✔
AUT: Notes	ϕ_H	Novel Bug	The user must be able to go to AboutActivity and return back.	(a)	✔
				(b)	✔
				(c)	✔
	ϕ_I	Bug Reports	The user must be able to go to SettingsActivity and return back.	(a)	✔
				(b)	✔
				(c)	✔

We downloaded two applications from F-Droid, namely ChessWalk and Notes. F-Droid [12] is an Android GUI application database, and many Android testing studies use it. We find F-Droid useful because it provides old versions and bug reports of the applications.

Table 6.2 lists the GUI-level test scenarios we obtained for ChessWalk and Notes applications. For more detailed information, LTL specifications of these test scenarios

are available in Chapter C. These specifications come from four sources, (i) app-agnostic test oracles [31], (ii) bug reports in the F-Droid database, (iii) novel bugs we found, and (iv) specifications we manually created. Note that we can specify the same GUI function with different LTL formulae. In an LTL formula, an action label starts with the word *action*. Otherwise, it is a state label. There are two kinds of action labels, type and detail. An action type label constrains the action type, while an action detail label constrains the action parameters. Using these label categories, we define three levels of detail for LTL formulae with (a) only state labels, (b) state labels and action type labels, and (c) all labels. Intuitively, FARLEAD should be more effective as the level of detail goes from (a) to (c). Note that specifications ϕ_C and ϕ_I are inexpressable with a level (a) formula because they explicitly depend on action labels.

We investigate FARLEAD in three categories, FARLEADa, FARLEADb, and FARLEADc, indicating that we use a level (a), (b), or (c) formula, respectively. For specifications ϕ_C and ϕ_D, the LTL formula does not change from level (b) to (c) because the specified action does not take any parameters. Hence, we combine FARLEADb and FARLEADc as FARLEADb/c for these specifications. Other than FARLEAD, we perform experiments on three known approaches, (i) random exploration (Random), (ii) Google's built-in monkey tester (Monkey) [25], and (iii) Q-Learning Based Exploration optimized for activity coverage (QBEa) [19]. Random explores the AUT with completely random actions using the same action set of FARLEAD. Monkey also explores the AUT randomly, but with its own action set. QBEa chooses actions according to a pre-learned probability distribution optimized for traversing activities. We implement these approaches in FARLEAD so we can check if they satisfy our specifications, on-the-fly.

For every test scenario in Table 6.2, we execute Random, Monkey, QBEa, FARLEADa, FARLEADb, and FARLEADc 100 times each for a maximum of $E = 500$ episodes. The maximum number of steps is $K = 4$ or $K = 6$, depending on the specification, so every execution runs up to 500 episodes with at most six steps per episode. We keep the remaining parameters of FARLEAD fixed throughout our experiments.

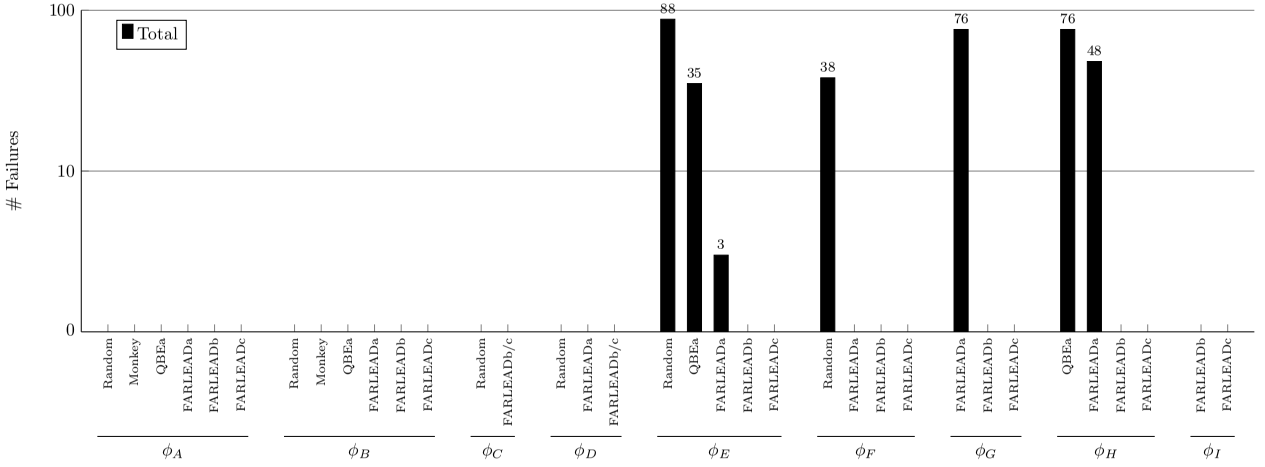
Table 6.3. Test Generator Effectiveness.

Engine	ϕ_A	ϕ_B	ϕ_C	ϕ_D	ϕ_E	ϕ_F	ϕ_G	ϕ_H	ϕ_I	**Total**
Random	✔	✔	✔	✔	✔	✔				6
Monkey	✔	✔					✔			3
QBEa	✔	✔			✔			✔		4
FARLEADa	✔	✔		✔	✔	✔	✔	✔		7
FARLEADb	✔	✔	✔	✔	✔	✔	✔	✔	✔	9
FARLEADc	✔	✔	✔	✔	✔	✔	✔	✔	✔	9

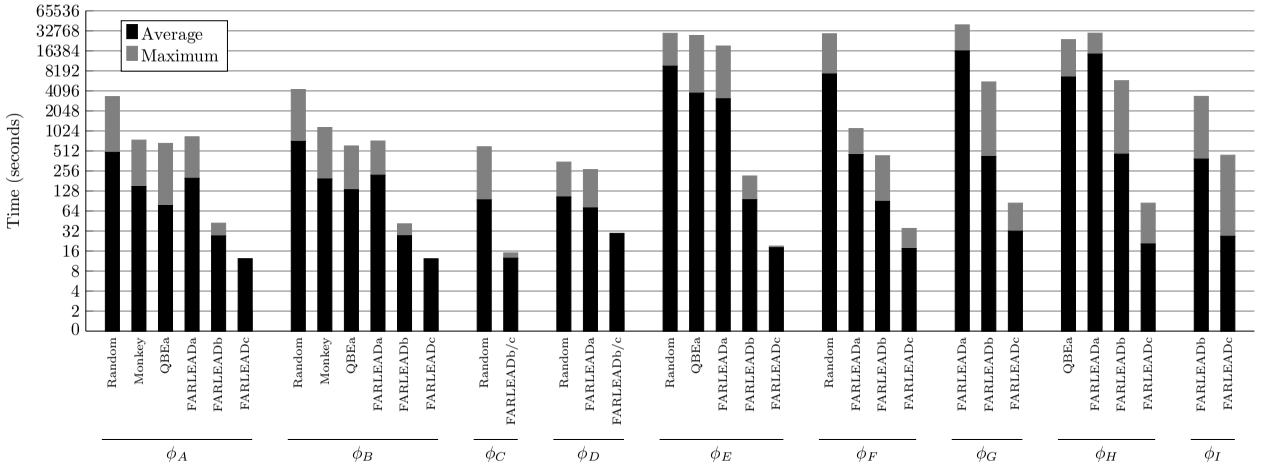
Table 6.3 shows the total number of test scenarios that a test generator were effective at satisfying. Our results show that FARLEADa, FARLEADb, and FARLEADc were effective at more specifications than Random, Monkey, and QBEa. Hence, we conclude that FARLEAD is the most effective functional test generator. Table 6.3 also shows that FARLEADb and FARLEADc were effective at more specifications than FARLEADa. Hence, we conclude that FARLEAD-Android becomes more effective when the level of detail goes from (a) to (b) or (c).

Figure 6.7 shows our experimental performance results. Figure 6.7(a) shows the number of failures of all the engines across 100 executions in logarithmic scale. According to this figure, FARLEADa, FARLEADb, and FARLEADc failed fewer times than Random, Monkey, and QBEa at ϕ_E, ϕ_F, and ϕ_H, indicating that FARLEAD achieved higher performance for these specifications. Only FARLEAD was effective at ϕ_G and ϕ_I, so we ignore those specifications in evaluating performance.

Figure 6.7(b) shows the average and the maximum times required to terminate for all the engines with every specification across 100 executions in logarithmic scale. According to this figure, FARLEADb and FARLEADc spent less time on average and in the worst case than Random, Monkey, and QBEa for the remaining test scenarios ϕ_A-ϕ_D, indicating that FARLEAD achieved higher performance when it used level (b) or (c).



(a) Number of Failures Across 100 Executions.



(b) Test Times Across 100 Executions.

Figure 6.7. Experimental Performance Results.

Figure 6.7(b) shows that QBEa and Monkey spent less time than FARLEADa for ϕ_A and ϕ_B because these test scenarios are about traversing activities only, a task which QBEa explicitly specializes on and Monkey excels at [5]. As a last note, FARLEADa outperformed QBEa for ϕ_H even though QBEa spent less time than FARLEADa because QBEa failed more than FARLEADa. Hence, we conclude that FARLEAD achieves higher performance than Random, Monkey, and QBEa unless the specification is at level (a) and about traversing activities only.

Figure 6.7(a) shows that FARLEADa failed more times than FARLEADb and FARLEADc at ϕ_E-ϕ_H, indicating FARLEADb and FARLEADc outperformed FARLEADa. FARLEADa was ineffective at ϕ_C and ϕ_I, so we ignore those specifications in evaluating performance. Figure 6.7(b) shows that FARLEADa spent more time than FARLEADb and FARLEADc at the remaining specifications, ϕ_A, ϕ_B, and ϕ_D, indicating that FARLEADb and FARLEADc again outperformed FARLEADa. Furthermore, Figure 6.7(b) shows that FARLEADc spent less time than FARLEADb in all specifications except ϕ_C and ϕ_D, where FARLEADb and FARLEADc are equivalent. Hence, we conclude that the performance of FARLEAD-Android increases as the level of detail goes from (a) to (c).

Overall, FARLEAD is the most effective test generator. It also outperforms the other test generators. Note that FARLEAD's effectiveness and performance improve with the level of detail in the test scenario description.

6.5. Notes on FARLEAD

Through dynamic execution, the RL agent learns from positive and negative rewards, on-the-fly for every action taken. Typically, an RL agent is trained to keep getting positive rewards and avoid the negative ones, indefinitely. Instead, our goal is to generate one satisfying test for a specified test oracle and terminate, which requires much less training than typical RL use cases. This helps us to develop a test generator with low execution costs, which is crucial for dynamic execution tools.

In addition to verifying GUI functions, note that FARLEAD also reproduces known GUI bugs. A GUI bug may have test scenario just as a GUI function. Hence, from the FARLEAD-Android perspective, a GUI bug is equivalent to a GUI function.

7. FARLEAD2: IMPROVING FARLEAD WITH EXPERIENCE REPLAY

Although FARLEAD is the first test generator enabling fully automated functional testing of Android GUI applications, our experience [23] shows it has two shortcomings:

(i) As an *online* learner, for every test scenario, even if the AUT is the same, FARLEAD starts learning from scratch, which is time consuming and repetitive.
(ii) The test scenario format FARLEAD accepts is not realistic because typical test scenarios are human-readable, natural language descriptions. Instead, FARLEAD scenarios are complicated formal specifications. The developer/tester has to learn LTL before generating tests with FARLEAD, which is impractical.

In a recent study, we replaced LTL scenarios with a common UI automation syntax called Gherkin [22]. However, Gherkin is not a complete language, it is just a syntax without semantics. Trying to fit a predefined semantics to Gherkin proved to be challenging, making many Gherkin keywords obsolete or unintuitive. Our experience shows that even with Gherkin syntax, writing and maintaining test scenarios remain difficult for the tester/developer

We propose FARLEAD2, an improvement to FARLEAD. We improve FARLEAD by enhancing its Reinforcement Learning (RL) algorithm with Generalized Experience Replay (GER). GER gathers experience while witnessing a test scenario and then utilizes that experience for later test scenarios. We also propose the Staged Test Scenario (STS), a human-readable but unambiguous test scenario language that divides the underlying tasks of a test scenario into consecutive stages. These stages produce intermediate positive rewards upon completion, enabling Reward Shaping. STSs also allow the developer/tester to easily incorporate apriori information on the AUT, facilitating witness generation.

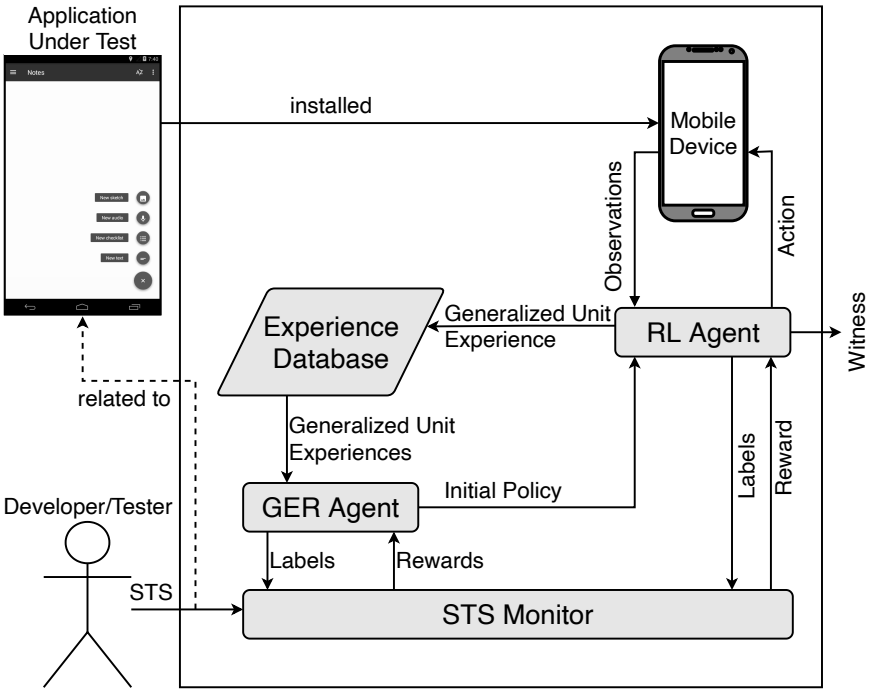


Figure 7.1. FARLEAD2 Overview.

Figure 7.1 shows the overview of FARLEAD2. First, FARLEAD2 assumes the AUT is installed on a Mobile Device. Second, the Developer/Tester provides an STS test scenario. The STS monitor receives this STS, then computes the monitor state and calculates the immediate Reward. Third, the GER Agent generates Labels (Boolean propositions) by processing every Generalized Unit Experience residing in the Experience Database, where a Generalized Unit Experience is a transition between two Device States via an Action. The GER Agent receives Rewards from the STS monitor and learns an Initial Policy. After exhausting all the Generalized Unit Experiences, the GER Agent sends the Initial Policy to the RL Agent. Finally, after receiving the Initial Policy, the RL Agent starts searching for a Witness by selecting an Action according to its current policy and executing it on the Mobile Device. Then, the RL Agent observes the Device State and generates a Generalized Unit Experience along with Labels, storing the Generalized Unit Experience in the Experience Database and sending the Labels back to the STS Monitor. The STS Monitor calculates a Reward from the Labels and sends it back to the RL Agent. The RL Agent learns from the Reward, in other words, updates its policy accordingly. FARLEAD2 continues searching until either it finds a Witness or gives up the search after a predefined limit.

FARLEAD2's main contributions to the literature are

(i) being the first study that combines RL and GER for GUI test generation,

(ii) the usage of novel STS, and

(iii) an experimental evaluation of RL with GER (RL+GER), showing that it fails fewer times, witnesses more test scenarios, and is faster than RaNDom (RND) and RL witness generators.

7.1. Staged Test Scenarios

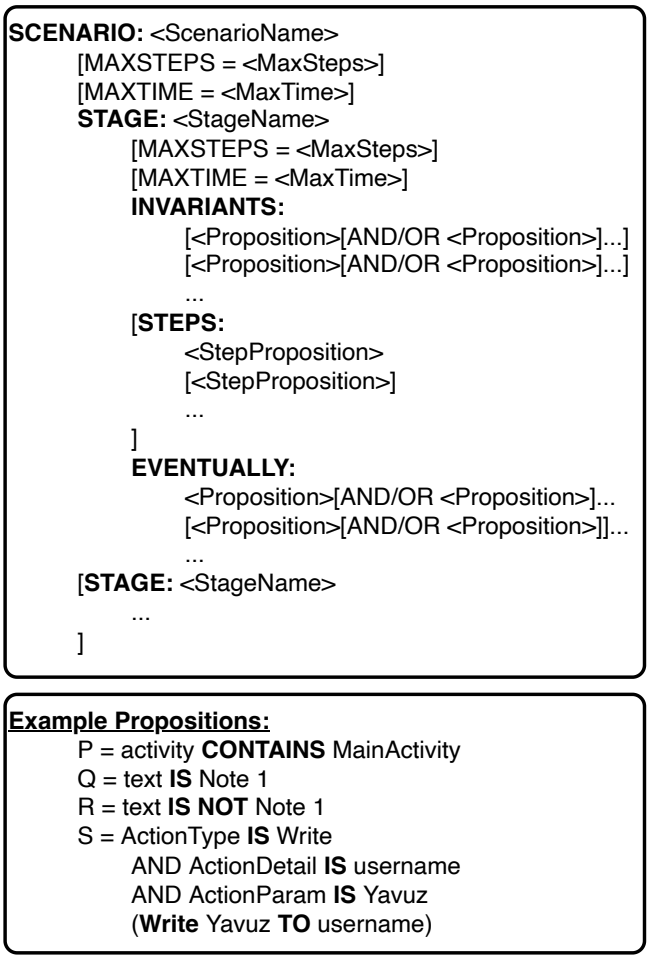


Figure 7.2. Staged Test Scenario (STS) Overview.

Figure 7.2 shows the overview of a Staged Test Scenario (STS). In the STS format, every word inside angle and square brackets are variables and optional constructs, respectively. Keywords separated with slashes are alternatives to each other.

Within given time constraints and step bounds, a candidate witnesses a stage only if all its invariants are true until all its eventual conditions become true. In other words, all invariants and all eventual conditions are necessary and sufficient propositions, respectively. Note that this is similar to the until operator in LTL. All invariants and eventual conditions are Boolean propositions. These propositions do not have to be atomic. They may have several terms connected via AND or OR operators.

We assume that every Boolean proposition is a triple (property, relation, value). There are two types of properties, action and state properties. There are three action property types. These are ActionType, ActionParam, and ActionDetail, linked to the GUI action type, the action parameter, and an attribute of the related widget, respectively. State properties are either crashed, package, activity, or one of any GUI widget's attributes on the screen. The relation is IS, IS NOT, CONTAINS, or NOT CONTAINS. Since every relation's negation (NOT) is available along with the AND and OR operators, our Boolean propositions are functionally complete. In other words, our Boolean operations are universal.

In Figure 7.2, P, Q, R, and S are some example propositions. These propositions become labels only if the current activity name contains MainActivity, the screen has a text that writes exactly "Note 1", there are no texts that write exactly "Note 1", and the GUI action types the word "Yavuz" to username, respectively.

A stage may have optional steps. These steps are a list of propositions with action properties only. For example, S is a step proposition, and P, Q, and R are not in Figure 7.2.

Step propositions can be cumbersome to specify. To address this issue, we have developed a shorthand notation. FARLEAD2 automatically converts every shorthand notation in the STS to a proper step proposition. Figure 7.2 gives the shorthand notation for S between round brackets.

7.1.1. Monitoring an STS Stage

Given an STS, FARLEAD2 aims to find the witness via trial-and-error, generating many candidate tests before the one that witnesses the STS. FARLEAD2 generates and executes one GUI action at every step. To learn after every step, FARLEAD2 maintains a reward variable, R. Most of the R values in FARLEAD2 are typical for an RL agent. When R = +1.00, the candidate at hand (the GUI action sequence up to now) is indeed a witness. When R = −1.00, the candidate will never become a witness due to its previous GUI actions. When R = 0.00, the candidate does get neither closer to nor farther away from being a witness. In addition to these typical values, FARLEAD2 may also have atypical partial reward values between 0.00 and +1.00. In this case, the candidate is not yet a witness but satisfies some of the conditions of becoming one. In the literature, using such partial reward values is known as Reward Shaping (RS) [24].

At every step, an STS monitor should calculate the reward value automatically by checking the currently monitored propositions at that step for consistency with the test scenario. All these propositions are Boolean. FARLEAD2 observes some of these propositions during one of the steps. All these observed propositions are labels of that step.

In a FARLEAD2 step, there are two types of propositions, necessary and sufficient. These propositions create three possibilities at every step.

(i) Sufficient propositions are a subset of the labels. In this case, the monitor returns a positive reward, a full plus one if this is the last step of the given test scenario.
(ii) Sufficient propositions are not a subset of the labels, and at least one necessary proposition is not a label. Then, the monitor returns a minus one reward.
(iii) In all other cases, the monitor returns zero.

Note that a proposition is related to either the current GUI action or GUI state, where a GUI state is all the GUI widgets' attributes on the screen.

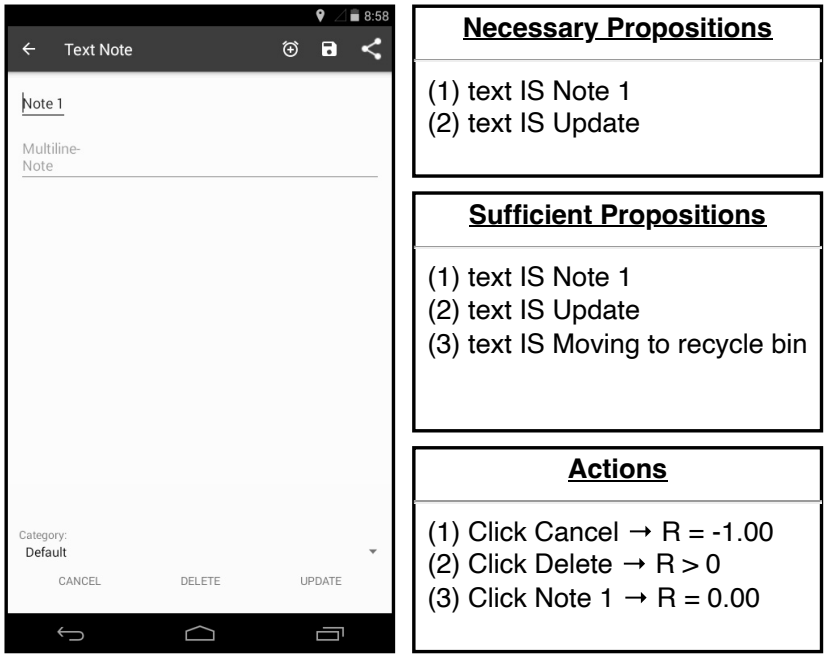


Figure 7.3. An Example Step of FARLEAD2.

Figure 7.3 illustrates an example step of FARLEAD2. On the screen to the left, all the necessary propositions are labels. However, the "text IS Moving to recycle bin" proposition is not. FARLEAD2 must find the correct GUI action, so this proposition also becomes a label, and FARLEAD2 gets a positive reward for that. In reality, there are many more GUI actions enabled on the screen to the left. But for the sake of simplicity, we consider only the Click Cancel, Delete, and Note 1 actions. Clicking Cancel closes the current screen. After this action, the monitor generates a minus one reward because (i) the "text IS Moving to recycle bin" proposition did not become a label, and (ii) the necessary propositions stopped being labels. Click Delete action opens a popup with "Moving to recycle bin" text without closing the current screen, making all the propositions labels. Therefore, the monitor generates a positive reward, a full plus one if this action witnesses the whole test scenario. Click Note 1 action clicks the text at the top, so the screen remains unchanged where still, the necessary propositions are labels, but the "text IS Moving to recycle bin" proposition did not become a label. Therefore, the monitor generates a zero reward. All propositions in this example are state propositions. Action propositions may constrain what actions FARLEAD2 should take. In that case, FARLEAD2 automatically reduces the set of enabled actions to avoid any future negative rewards and pursue positive ones.

7.2. Generalized Experience Replay (GER)

Every unit experience (s, a, s', r) in traditional Experience Replay (ER) has a fixed reward r. However, after FARLEAD2 witnessing a scenario, reward values will never be the same because of the following reasoning. Once FARLEAD2 witnesses a test scenario, the developer/tester will never use FARLEAD2 again to verify the related GUI function since the developer/tester already has a replayable witness for it. Therefore, the developer/tester will always use FARLEAD2 with unique test scenarios (scenarios FARLEAD2 has never witnessed before). Every unique test scenario yields a different reward function. Hence, if FARLEAD2 uses traditional ER, some of the recorded rewards are bound to be misleading for the new test scenario, hampering witness generation effectiveness and performance.

The FARLEAD2 monitor generates a reward value at every step by checking the labels of that step. FARLEAD2 determines these labels by looking at the step's GUI action a and the GUI state s' reached after executing that GUI action. In other words, the reward r is always a function of the GUI action a and the GUI state s'. Hence, storing only GUI states and GUI actions is sufficient to calculate reward values for any test scenario.

A generalized unit experience is a triple (s, a, s'), meaning that the AUT goes from state s to state s' by executing action a. We call it generalized because we generalize the reward value out. Note that storing only (a, s') would be sufficient to calculate the reward value, but it would be insufficient to determine which state-action pair (s, a) gets that value.

Figure 7.4 demonstrates an example in which the generalized experience gathered in a test scenario facilitates witnessing a second. These scenarios are about reaching different screens of the AUT, both in two steps. We already have a witness for the first test scenario. This witness has two GUI actions, A and B. For the second test scenario, we do not have a witness yet. So, the GUI actions C and D are unknown. Before any

exploration, the GER module replays the generalized experience gathered from the first witness. During replay, the GUI action A gets a positive reward value because it is consistent with the first step of the second test scenario. However, the GER module assigns a negative reward value for the GUI action B because it is inconsistent with the second step of the test scenario. As a result, FARLEAD2 picks C=A with no exploration and eliminates B as a candidate for the second step. Overall, the search space for the second witness shrinks, amplifying FARLEAD2 effectiveness and performance.

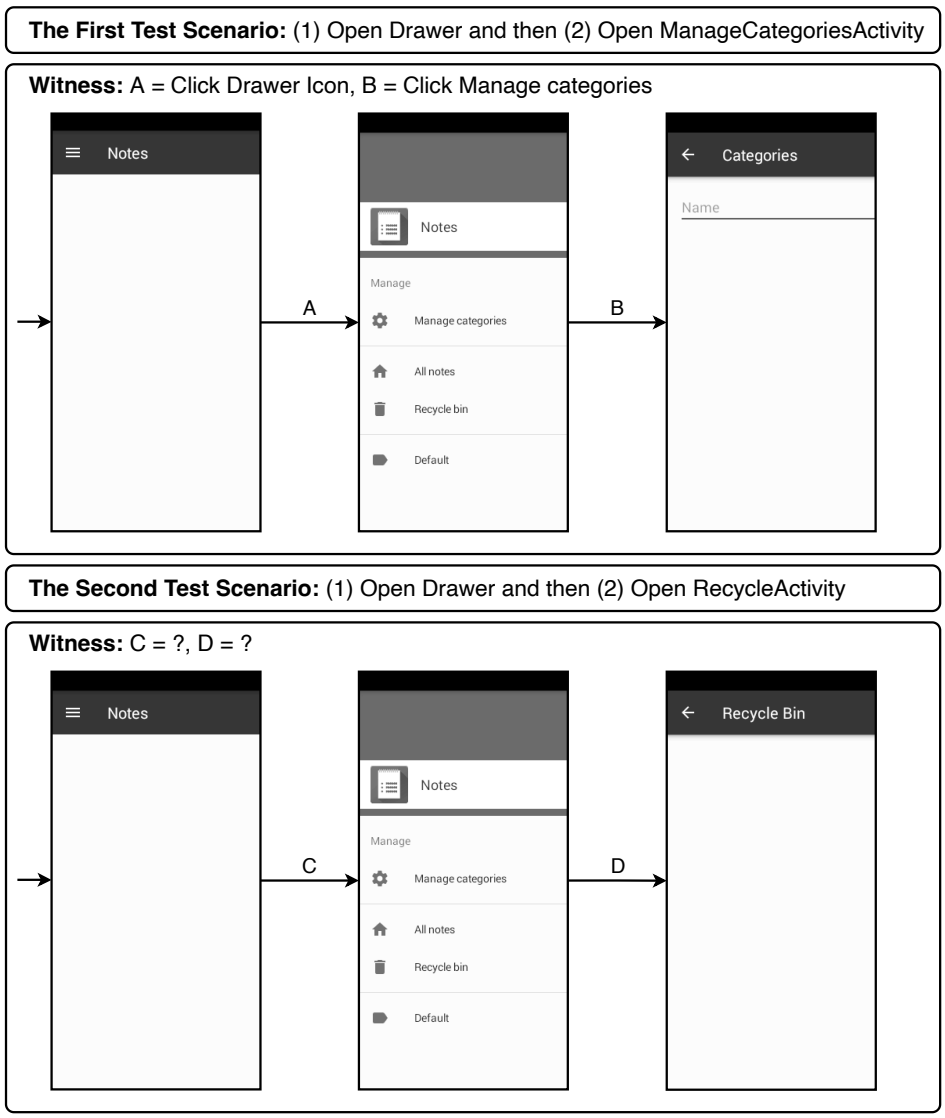


Figure 7.4. Generalized Experience Replay (GER) Example.

As a final note, a FARLEAD2 state is different than a typical GUI state, because it is a combination of the GUI state and the STS monitor state. The STS monitor

starts from an initial state m_0, denoting it is monitoring the first stage. If the first stage is complete, the monitor goes to state m_1. Hence, the state subscript denotes the last completed stage.

7.3. Evaluation

This section describes our experimental setup, discusses our research questions, and evaluates our experimental results.

7.3.1. Experimental Setup

7.3.1.1. Witness Generators. We compare three witness generators, RaNDom (RND), Reinforcement Learning (RL), and Reinforcement Learning with Generalized Experience Replay (RL+GER). RND generates random GUI actions, ignoring all rewards. RL uses Reinforcement Learning (RL) to obtain the witness. It is equivalent to FARLEAD-Android but with STS monitoring instead of LTL. Finally, RL+GER uses RL with Generalized Experience Replay (GER).

7.3.1.2. Effectiveness. A witness generator fails to produce a witness only if it hits its episode limit, which is 100 throughout our experiments. Otherwise, it outputs a witness and thereby is successful. A witness generator's effectiveness is the percentage of times it is successful. We execute the same witness generator for the same test scenario under the same conditions ten times. So, the witness generator is a hundred percent effective if it generates a witness ten times. Conversely, it is zero percent effective if it fails all the time. Higher effectiveness directly shows that the witness generator fails fewer times.

7.3.1.3. Performance. A witness generator performs better than another if it terminates faster. We have two measures reflecting performance, (i) total number of steps and (ii) total seconds it takes until termination. We look at the first measurement

to ensure the latter does not suffer from noise caused by varying execution times of individual GUI actions on the mobile device. Since we execute the same scenario under the same conditions ten times, we take the average of both performance measures.

7.3.1.4. The Mobile Device. Throughout our experiments, the mobile device is a VirtualBox guest with 1024 megabytes of random access memory and a screen resolution of 480x800. The operating system of this device is an Intel x86 port of Android 4.4.5. Using a VirtualBox guest allows making exact clones of our experimental environment, allowing mass witness generation for different test scenarios in parallel. Furthermore, no physical mobile devices or hardware preparation are required to replicate our experiments.

7.3.1.5. Application Under Test (AUT). Throughout our experiments, the Application Under Test (AUT) is the "org.secuso.privacyfriendlynotes" package from F-Droid [12], Notes in short. It allows the user to create four types of notes; audio, text, sketch, and checklist. Furthermore, the user may construct categories and divide notes into those categories.

The Notes application has a known bug in its sketch notes where the color palette has no black color, preventing the user from making black drawings [73].

7.3.1.6. Test Scenarios. For the Notes application, our experimental setup has 17 test scenarios in the order of increasing complexity. We measure the complexity of a test scenario as the length of its shortest witness. The shortest witness length is the minimum number of steps (GUI actions) required to witness the test scenario. We argue that a witness generator would have more difficulties in a complex test scenario due to the number of unknown steps it needs to discover.

The complexities of our experimental test scenarios vary between 2 and 13. Figure 7.5 shows an example witness manually generated for test scenario 014. The existence

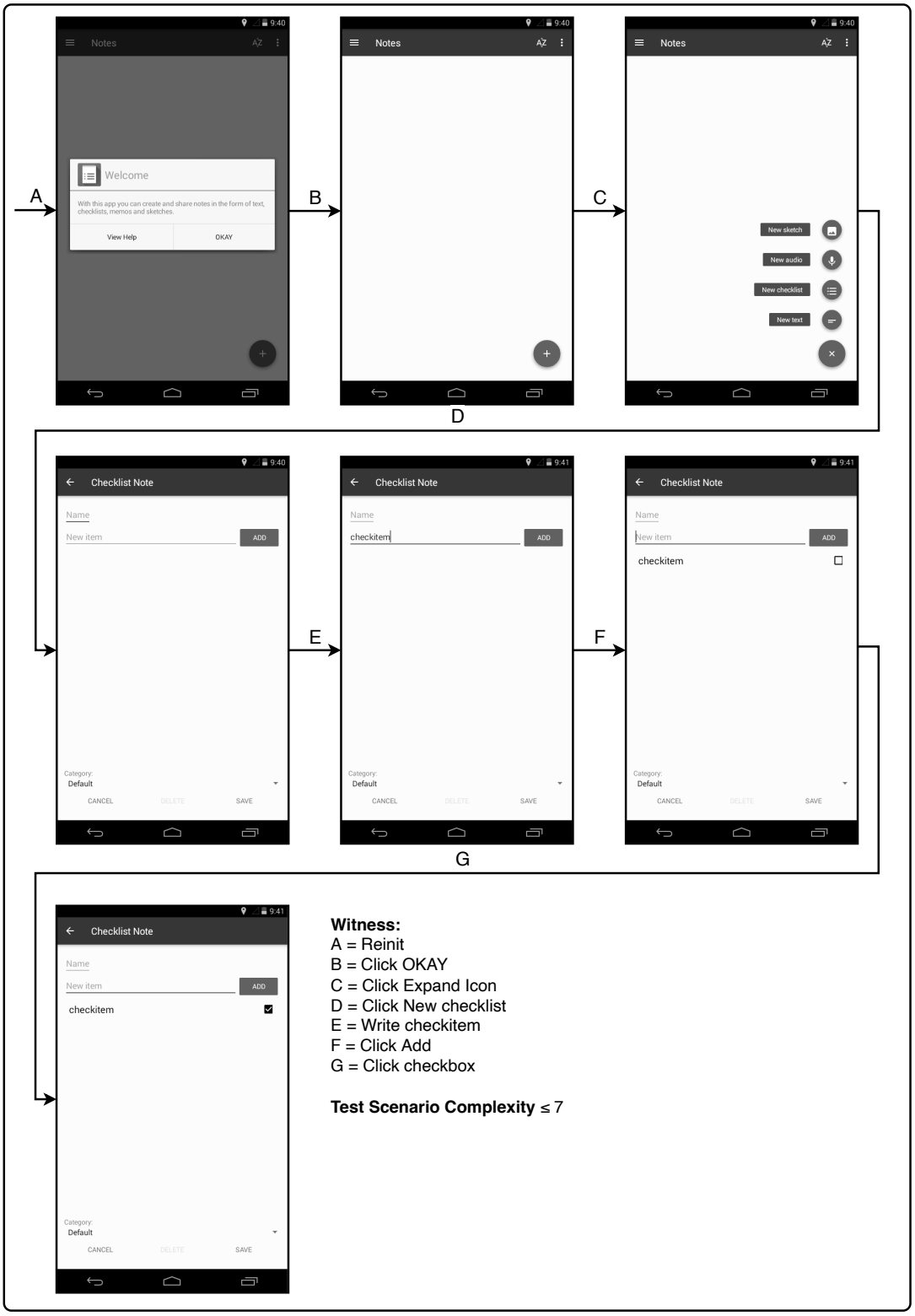


Figure 7.5. A Witness for Test Scenario 014.

of this witness puts an upper bound of 7 on the complexity of this test scenario. We manually produce witnesses for all test scenarios to determine their complexity.

Staged Test Scenario (STS) is a flexible structure, allowing the developer/tester to incorporate apriori information about the test scenario. According to the level of information given to an STS, there are two extremes. These are declarative and imperative STSs.

A declarative STS contains only the necessary information for a scenario. This information is (i) the invariants and (ii) the eventual conditions of every stage. So, the developer/tester declares only what the generator should witness. In contrast, an imperative STS defines the steps of every stage. An imperative STS shrinks the search space, so there is often only one candidate. However, an imperative STS is cumbersome to maintain because it requires restructuring after almost any software update, whereas a declarative STS should work across multiple versions of the AUT.

For every experimental test scenario, we have four STSs, with four levels of information; L4 (imperative), L3, L2, and L1 (declarative). Hence, for the 17 test scenarios, we get 68 STSs in total.

Figure 7.6 shows L1-L4 STSs for the test scenario 014. The first stage of L1 has no invariants but only one eventual condition, starting the AUT package on the device. The second stage has one invariant, describing that the AUT package must be active until the second stage's eventual condition is satisfied, so the ChecklistNoteActivity is on the screen. Again, the third stage has the same invariant, describing that the AUT package must be active, but now it is until the device ends up in the ChecklistNote-Activity, while there is a text that writes "checkitem" and there is a checked checkbox on the screen. Overall, the L1-STS describes that (i) eventually, the AUT must be opened. (ii) Then, eventually, the ChecklistNoteActivity must be opened. (iii) Finally, the ChecklistNoteActivity must be on the screen with the checklist containing a checked item, and the "checkitem" text appears on the screen. Whenever FARLEAD2

SCENARIO: 014_L1
STAGE: OpenNotes
EVENTUALLY:
package **IS** org.secuso.privacyfriendlynotes
STAGE: GoToChecklistNoteActivity
INVARIANTS:
package **IS** org.secuso.privacyfriendlynotes
EVENTUALLY:
activity **CONTAINS** ChecklistNoteActivity
STAGE: CreateItemAndCheck
INVARIANTS:
package **IS** org.secuso.privacyfriendlynotes
EVENTUALLY:
activity **CONTAINS** ChecklistNoteActivity
text **IS** checkitem
checked **IS** true

SCENARIO: 014_L2
STAGE: OpenNotesAndGoToMainActivity
EVENTUALLY:
package **IS** org.secuso.privacyfriendlynotes
activity **CONTAINS** MainActivity
STAGE: GoToChecklistNoteActivity
INVARIANTS:
package **IS** org.secuso.privacyfriendlynotes
activity **CONTAINS** MainActivity
EVENTUALLY:
activity **CONTAINS** ChecklistNoteActivity
STAGE: CreateItemAndCheck
INVARIANTS:
package **IS** org.secuso.privacyfriendlynotes
activity **CONTAINS** ChecklistNoteActivity
EVENTUALLY:
activity **CONTAINS** ChecklistNoteActivity
text **IS** checkitem
checked **IS** true

SCENARIO: 014_L3
STAGE: OpenNotesAndGoToMainActivity
MAXSTEPS = 1
EVENTUALLY:
package **IS** org.secuso.privacyfriendlynotes
activity **CONTAINS** MainActivity
STAGE: ExpandDrawer
MAXSTEPS = 2
INVARIANTS:
actionType **IS** click
package **IS** org.secuso.privacyfriendlynotes
activity **CONTAINS** MainActivity
EVENTUALLY:
text **CONTAINS** New
activity **CONTAINS** MainActivity
STAGE: GoToChecklistNoteActivity
MAXSTEPS = 1
INVARIANTS:
actionType **IS** click
text **CONTAINS** New
package **IS** org.secuso.privacyfriendlynotes
activity **CONTAINS** MainActivity
EVENTUALLY:
activity **CONTAINS** ChecklistNoteActivity
STAGE: CreateItemAndCheck
MAXSTEPS = 3
INVARIANTS:
actionType **IS** click **OR** actionType **IS** write
activity **CONTAINS** ChecklistNoteActivity
EVENTUALLY:
activity **CONTAINS** ChecklistNoteActivity
text **IS** checkitem
checked **IS** true

SCENARIO: 014_L4
STAGE: OpenNotesAndGoToMainActivity
STEPS:
reinit
EVENTUALLY:
package **IS** org.secuso.privacyfriendlynotes
activity **CONTAINS** MainActivity
STAGE: GoToChecklistNoteActivity
INVARIANTS:
package **IS** org.secuso.privacyfriendlynotes
STEPS:
click OKAY
click expand
click checklist
EVENTUALLY:
activity **CONTAINS** ChecklistNoteActivity
STAGE: CreateItemAndCheck
INVARIANTS:
package **IS** org.secuso.privacyfriendlynotes
STEPS:
write checkitem
click Add
click checkitem
EVENTUALLY:
activity **CONTAINS** ChecklistNoteActivity
text **IS** checkitem
checked **IS** true

Figure 7.6. L1-L4 STSs for Test Scenario 014.

encounters a text proposition, note that it automatically considers writing that text to any appropriate GUI widget an enabled action.

Given an L1-STS, we automatically generate an L2-STS through intent resolution analysis [14]. Intent resolution analysis is a static analysis on an Android GUI application binary that extracts the Static Activity Transition Graph (SATG) of an AUT. The SATG determines from any activity, which activities a tester can go to, and using the SATG, FARLEAD2 updates the given STS with extra invariants and eventual conditions. For the test scenario 014, FARLEAD2 determines that it can reach ChecklistNoteActivity via MainActivity. So, it automatically restricts its search to those activities by adding activity constraints to appropriate stages of the STS. Overall, the L2-STS shrinks the search space with no manual effort.

The L3-STS incorporates any information the developer/tester may give, except the steps (GUI actions) themselves. First, MAXSTEPS defines the maximum number of steps allowed for every stage. Note that assigning every MAXSTEPS condition to an absolute minimum would force the witness generator to find the shortest witness, taking more time than finding an arbitrary witness. So, we put slightly relaxed values in these conditions. Second, "actionType" propositions restrict the action type. Third, an extra stage called ExpandDrawer describes that the text "New" must appear on the screen before reaching the ChecklistNoteActivity. All these additions shrink the search space but require extra manual effort.

The L4-STS is imperative and describes all the steps, so almost no learning is required. Figure 7.5 shows that after the GUI action D, FARLEAD2 still has to learn the correct GUI widget to write on. With the L4-STS, search space is the smallest, but the developer/tester determines all the GUI actions manually, making the manual effort of writing an L4-STS the highest among all STSs. Still, writing an executable test script requires coding skills, whereas an L4-STS is a no-code script. So, writing an L4-STS takes less effort than writing the test script itself.

We have designed one test scenario (4 STSs) to reach each activity of the Notes application (test scenarios 001-009). Hence, witnessing all the test scenarios achieve full activity coverage. We have created one test scenario (test scenario 012) to reproduce the palette bug [73]. Finally, the rest of the test scenarios are about the main GUI functions of the Notes application, namely, creating, deleting, recycling, and categorizing notes.

7.3.1.7. Generalized Experience Replay (GER) Setup. Generalized Experience Replay (GER) depends on the experience gathered so far. Our experimental setup starts with no experience and executes RL+GER on the test scenarios in the order of increasing test complexity. Even though RL+GER re-witnesses an STS multiple times in our experiments, it never uses the experience of that same STS. RL+GER selects one run per previous STS and uses the cumulative experience gathered from only these STSs. Hence, we expect RL+GER to produce similar results to RL in test scenario 001. Also, RL+GER will have the most experience when it witnesses the most complex test scenario (test scenario 017). Note that we use separate experience databases for every STS level. Finally, although we perform our experiments in parallel, RL+GER waits for the previous scenarios to finish before generating witnesses.

7.3.1.8. Overall. Our experimental setup has three witness generators (RND, RL, and RL+GER), 68 STSs, and ten runs for each witness generator-STS combination. Hence, there are 2040 experimental runs in total. We measure four values for every experimental run; (i) success/fail, (ii) the total number of steps, (iii) the total seconds, and (iv) the witness length. The first value measures effectiveness, the second and third values measure performance, and the last value measures test complexity.

7.3.2. Research Questions

Our experimental setup aims to answer the following research questions.

RQ1. (Feasibility) Are all the experimental test scenarios witnessable?

RQ2. (Effectiveness) How much more effective is RL+GER than RL and RND?

RQ3. (Performance) How much performance increase does RL+GER have over RL and RND?

RQ4. (Witness Length) How much longer witnesses RL+GER does generate over RL and RND?

RQ5. (Levels of Information) What is the impact of L1-L4 STSs on witness generation effectiveness and performance?

RQ6. (Test Scenario Complexity) How does test scenario complexity affect witness generation effectiveness and performance?

RQ1 verifies that every experimental test scenario has positive utility in evaluating effectiveness, performance, and test complexity. If the underlying GUI function that a test scenario exploits is nonexistent in the AUT, there exists no witness for that test scenario. Then, effectiveness will be zero percent regardless of the witness generator, and performance and test complexity measurements would be infeasible. We aim to show that there exists at least one witness for every experimental test scenario.

RQ2 evaluates the most crucial criterion for a witness generator, its effectiveness. Depending on the test scenario, an ineffective witness generator would often fail in practice, frustrating the developer/tester. We aim to ensure that RL+GER is more effective than RND and RL.

RQ3 evaluates how fast a witness generator terminates. A faster and more effective witness generator would produce more witnesses within a constant testing budget, providing the developer/tester more utility. We aim to show that RL+GER outperforms RND and RL.

RQ4 evaluates the increase in average test complexity due to high effectiveness. A highly effective witness generator finds witnesses for complex test scenarios, test scenarios with longer witnesses by definition. We aim to measure the re-execution cost of a witness suite created by RL+GER over RND and RL. From the developer/tester

perspective, re-execution costs are relevant in the case of regression testing. Regression testing involves re-executing a previous test suite on a new software version, and it is imperative to finish re-execution as fast as possible.

RQ5 aims to determine RL+GER's effectiveness and performance under different levels of information. Our goal is to show that RL+GER is preferable regardless of the information level.

Finally, RQ6 evaluates the effects of increasing text complexity over witness generation performance and effectiveness. Our experience shows that real-world test scenarios are complex enough to cause effectiveness and performance problems. Hence, we aim to demonstrate that RL+GER is more robust to increasing test complexity than RND and RL.

7.3.3. Experimental Results

Table 7.1 shows our experimental results. Every row in this table shows (i) effectiveness as a percentage, (ii) the total number of steps and total time required to generate a witness (performance), or (iii) the witness length (test complexity). Hence, every test scenario (001-017) has three rows, effectiveness, performance, and test complexity. The three rows on top show the average measurements across all test scenarios.

The three groups of columns RND, RL, and RL+GER of Table 7.1 show the measurements under L4-L1 STSs. The rightmost three columns show the average measures across all levels of information. The three by three group of entries at the top right corner of this table are the averages across all test scenarios and information levels.

Table 7.1. Experimental Results.

		RND				RL				RL+GER				Overall		
		L4	**L3**	**L2**	**L1**	**L4**	**L3**	**L2**	**L1**	**L4**	**L3**	**L2**	**L1**	**RND**	**RL**	**RL+GER**
All	**Effectiveness (%)**	100	57.6	45.3	62.9	100	80.0	93.5	74.7	100	98.2	96.5	82.4	**66.5**	**87.1**	**94.3**
	# Steps/Time (sec)	6.02/20.8	202/734	236/826	317/1021	5.86/19.9	214/823	243/818	490/1452	5.78/20.2	95.7/330	176/630	392/1240	**190/650**	**238/778**	**167/555**
	Witness Length	5.53	4.79	5.05	6.10	5.53	5.33	7.29	6.47	5.53	6.22	7.38	6.77	**5.42**	**6.16**	**6.45**
001	**Effectiveness (%)**	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
	# Steps/Time (sec)	2.00/6.90	4.20/15.9	10.5/37.5	13.7/47.7	2.00/7.20	2.80/10.5	4.40/12.6	7.40/26.2	2.00/7.50	4.00/15.2	9.10/30.8	10.9/40.1	7.60 / 27.0	4.15 / 14.1	6.50 / 23.4
	Witness Length	2.00	2.60	2.90	2.70	2.00	2.00	2.30	2.00	2.00	2.00	2.10	3.20	2.55	2.08	2.33
002	**Effectiveness (%)**	100	100	40.0	90.0	100	100	100	100	100	100	100	90.0	82.5	100	97.5
	# Steps/Time (sec)	4.00/16.2	163/597	255/899	267/860	4.00/16.0	29.4/115	146/505	211/673	4.00/16.3	21.4/84.7	118/413	316/963	172 / 593	97.7 / 327	115 / 369
	Witness Length	4.00	5.20	5.00	6.22	4.00	4.50	5.40	5.20	4.00	4.00	6.20	6.67	5.09	4.78	5.18
003	**Effectiveness (%)**	100	90.0	70.0	90.0	100	100	100	90.0	100	100	100	100	87.5	97.5	100
	# Steps/Time (sec)	4.00/15.7	114/417	207/729	213/693	4.00/16.1	28.7/115	116/398	275/831	4.00/16.2	8.00/31.9	79.8/289	109/360	134 / 464	106 / 340	50.3 / 174
	Witness Length	4.00	4.44	5.29	5.56	4.00	4.20	5.90	5.78	4.00	4.50	5.70	5.20	4.77	4.95	4.85
004	**Effectiveness (%)**	100	100	80.0	90.0	100	100	100	100	100	100	100	100	92.5	100	100
	# Steps/Time (sec)	4.00/15.5	82.4/302	165/590	311/1025	4.00/15.9	29.3/114	108/369	206/642	4.00/16.9	11.7/45.4	58.3/206	61.0/214	140 / 483	86.7 / 285	33.8 / 121
	Witness Length	4.00	4.30	5.12	5.33	4.00	4.80	5.30	6.00	4.00	4.30	5.30	6.30	4.65	5.03	4.97
005	**Effectiveness (%)**	100	100	60.0	100	100	100	100	90.0	100	100	100	100	90.0	97.5	100
	# Steps/Time (sec)	4.00/15.9	120/446	178/622	230/749	4.00/15.5	34.8/133	152/524	306/906	4.00/16.0	7.10/28.4	62.0/223	125/382	133 / 458	124 / 394	49.5 / 162
	Witness Length	4.00	4.70	5.50	6.20	4.00	4.60	5.60	5.56	4.00	4.40	5.80	5.90	5.06	4.92	5.03
006	**Effectiveness (%)**	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
	# Steps/Time (sec)	4.00/13.3	31.2/98.9	63.1/221	73.9/241	4.00/13.0	14.3/46.2	41.4/143	59.0/185	4.00/14.0	8.80/29.7	17.9/63.9	18.8/65.6	43.1 / 144	29.7 / 96.8	12.4 / 43.3
	Witness Length	4.00	5.50	5.20	6.00	4.00	4.30	5.60	7.20	4.00	4.70	4.40	5.90	5.17	5.28	4.75
007	**Effectiveness (%)**	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
	# Steps/Time (sec)	4.00/13.3	36.5/120	81.0/291	59.1/193	4.00/13.3	13.6/44.2	42.7/146	43.7/139	4.00/13.8	5.30/17.2	16.4/59.1	13.3/42.5	45.1 / 154	26.0 / 85.6	9.75 / 33.1
	Witness Length	4.00	5.30	4.50	6.10	4.00	4.60	5.00	5.00	4.00	4.20	5.10	5.60	4.97	4.65	4.72
008	**Effectiveness (%)**	100	90.0	50.0	80.0	100	100	100	100	100	100	100	100	80.0	100	100
	# Steps/Time (sec)	4.00/13.0	165/602	210/738	288/914	4.00/13.3	39.7/141	157/522	313/948	4.00/13.6	20.1/71.9	23.5/81.9	51.3/173	167 / 567	129 / 406	24.7 / 85.1
	Witness Length	4.00	4.89	6.00	5.38	4.00	4.80	6.50	6.00	4.00	4.90	5.80	6.00	4.91	5.33	5.17
009	**Effectiveness (%)**	100	80.0	70.0	100	100	100	100	100	100	100	100	100	87.5	100	100
	# Steps/Time (sec)	4.00/12.7	152/560	171/599	161/522	4.00/13.0	33.1/118	119/403	452/1341	4.00/13.4	15.0/52.7	25.7/93.7	38.9/118	122 / 423	152 / 469	20.9 / 69.5
	Witness Length	4.00	4.25	5.29	5.60	4.00	4.60	5.70	6.30	4.00	4.60	5.00	4.40	4.77	5.15	4.50
010	**Effectiveness (%)**	100	20.0	20.0	60.0	100	100	100	100	100	100	100	100	50.0	100	100
	# Steps/Time (sec)	5.00/17.4	301/1098	322/1142	376/1208	5.00/16.8	49.4/187	189/651	358/1049	5.00/17.9	17.6/64.4	32.6/115	85.7/263	251 / 866	150 / 476	35.2 / 115
	Witness Length	5.00	6.00	6.00	9.17	5.00	5.30	7.10	7.50	5.00	5.50	6.70	6.50	6.45	6.22	5.92
011	**Effectiveness (%)**	100	50.0	70.0	90.0	100	100	100	100	100	100	100	100	77.5	100	100
	# Steps/Time (sec)	5.00/17.7	246/894	239/838	315/1023	5.00/17.3	45.2/169	193/649	339/1036	5.00/18.0	10.8/39.9	30.6/113	66.4/200	201 / 693	145 / 468	28.2 / 92.8
	Witness Length	5.00	6.20	6.29	7.22	5.00	5.40	8.40	7.30	5.00	5.40	6.10	7.80	6.13	6.53	6.08
012	**Effectiveness (%)**	100	30.0	0.00	20.0	100	100	100	40.0	100	100	100	70.0	37.5	85.0	92.5
	# Steps/Time (sec)	5.00/17.6	292/1043	351/1230	497/1596	5.00/17.3	52.1/179	164/552	760/2350	5.00/17.7	10.1/36.0	35.5/121	476/1430	286 / 972	245 / 775	132 / 401
	Witness Length	5.00	6.33	-	8.00	5.00	5.80	6.20	8.25	5.00	5.30	6.50	6.71	5.67	5.97	5.81
013	**Effectiveness (%)**	100	0.00	0.00	10.0	100	0.00	100	20.0	100	100	100	70.0	27.5	55.0	92.5
	# Steps/Time (sec)	6.00/23.7	343/1265	350/1232	519/1641	6.00/23.6	775/3021	170/568	1251/3561	6.00/24.1	26.4/112	70.1/254	1174/3518	304 / 1040	550 / 1793	319 / 977
	Witness Length	6.00	-	-	13.0	6.00	-	12.3	15.5	6.00	8.70	14.0	12.6	6.64	9.73	10.1
014	**Effectiveness (%)**	100	0.00	0.00	0.00	100	0.00	40.0	0.00	100	100	70.0	0.00	25.0	35.0	67.5
	# Steps/Time (sec)	15.4/65.8	352/1286	352/1222	545/1755	12.6/51.9	1011/4415	685/2384	1096/3289	11.2/46.7	335/1491	847/3205	1344/4202	316 / 1082	701 / 2535	634 / 2236
	Witness Length	7.00	-	-	-	7.00	-	13.5	-	7.00	10.6	14.6	-	7.00	8.86	10.3
015	**Effectiveness (%)**	100	20.0	10.0	40.0	100	100	100	100	100	100	100	100	42.5	100	100
	# Steps/Time (sec)	8.00/22.6	333/1214	350/1227	428/1387	8.00/22.8	65.9/225	129/430	244/738	8.00/23.9	26.4/78.0	46.9/162	37.1/108	280 / 963	112 / 354	29.6 / 92.9
	Witness Length	8.00	6.50	9.00	10.2	8.00	8.10	6.80	8.50	8.00	7.50	7.80	8.70	8.41	7.85	8.00
016	**Effectiveness (%)**	100	0.00	0.00	0.00	100	0.00	70.0	0.00	100	100	70.0	0.00	25.0	42.5	67.5
	# Steps/Time (sec)	11.0/35.7	352/1266	351/1216	541/1730	11.0/34.8	785/3091	946/3312	1386/4069	11.0/35.8	472/1553	1155/4181	1842/6421	314 / 1062	782 / 2626	870 / 3048
	Witness Length	11.0	-	-	-	11.0	-	13.9	-	11.0	13.1	13.4	-	11.0	12.2	12.4
017	**Effectiveness (%)**	100	0.00	0.00	0.00	100	60.0	80.0	30.0	100	70.0	100	70.0	25.0	67.5	85.0
	# Steps/Time (sec)	13.0/30.2	347/1248	352/1212	560/1777	13.0/30.0	627/1871	772/2344	1015/2899	13.0/31.1	627/1862	364/1099	889/2586	318 / 1067	607 / 1786	473 / 1395
	Witness Length	13.0	-	-	-	13.0	15.8	15.9	15.3	13.0	14.6	14.9	14.0	13.0	14.7	14.1

We first calculate the effectiveness of a witness generator under an information level and a test scenario by counting the number of times the witness generator was successful and dividing by the total number of executions, which is ten. We multiply the result by a hundred to report it as a percentage. Second, every performance measure is an average of the ten executions. Finally, witness length is not applicable in case of a witness generator fails, so we take the average witness length of successful runs. If the witness generator failed all the ten executions, and the effectiveness is zero, we denote the witness length with a dash (-) character.

7.3.3.1. RQ1: Feasibility. Table 7.1 shows that under information level L4, all witness generators produced a witness for every test scenario with 100 percent effectiveness. Therefore, all the experimental test scenarios are feasible.

Note that an L4-STS delegates the task of finding low-level test steps to the developer/tester. Due to the resulting manual effort, the developer/tester would prefer lower information levels. At least, L4-STSs prove that a witness exists in the candidate space for every experimental test scenario.

7.3.3.2. RQ2: Effectiveness. Table 7.1 shows that, overall, RL+GER has 94.3 percent effectiveness. Across all test scenarios and information levels, RL+GER is 7.2 and 27.8 percent more effective than RL and RND, respectively. Furthermore, RL+GER has zero effectiveness in only two STSs, whereas RL and RND in five and twelve STSs, respectively. The lowest overall effectiveness of RL+GER is 67.5 (for test scenarios 014 and 016), where RL and RND's minimums are 35 and 25 (for test scenario 014), respectively. Therefore, RL+GER is consistently more effective than its alternatives and generates witnesses for more test scenarios.

The only case where RL+GER does not have the best effectiveness score across all information levels is test scenario 002. For this test scenario, RL beats RL+GER by 100 percent versus 97.5. The downside of GER is the rare occasion of maximization

bias due to false-lead experience. This bias occurs when the previous experience guides RL+GER to rapidly complete the first STS stages but in a way that makes it impossible to witness the whole STS. RL does not suffer from false leads because it does not replay any experience. Overall effectiveness scores show that the benefits of RL+GER outweigh the danger of false-lead experience maximization bias.

7.3.3.3. RQ3: Performance. Table 7.1 shows that, on average, across all test scenarios and information levels, RL+GER generates a witness in 167 steps and 555 seconds. RL+GER witness production is 71 steps and 223 seconds faster than RL. Again, it is 23 steps and 95 seconds faster than RND. These results show that RL+GER outperforms both RND and RL. RL+GER produces more witnesses within a constant testing budget because it is also the most effective witness generator.

Overall performance scores for each test scenario show that RL+GER outperforms both RND and RL in test scenarios 003-012 and 015. All witness generators witness the test scenario 002 in under half a minute, so the differences between witness generators are not significant. For test scenarios 013, 014, 016, and 017, RND outperforms both RL and RL+GER. A more detailed analysis of Table 2 shows that RND spends between 1212-1777 seconds for STSs with zero effectiveness, whereas RL and RL+GER spend between 3021-4069 and 4202-6421 seconds, respectively. For these complex test scenarios, RND terminates in failure significantly faster than RL and RL+GER. All our experimental witness generators start a new episode at the moment of any inconsistency between the generated candidate and the monitored test scenario. RND starts a new episode quicker than RL and RL+GER because it hits an inconsistency earlier. Hence, RND hits the episode limit (a hundred) faster than RL and RL+GER, allowing it to outperform RL but not RL+GER.

7.3.3.4. RQ4: Witness Length. Table 7.1 shows that, overall, RL+GER generates witnesses 6.45 steps long. Witnesses of RL+GER are 0.29 and 1.03 longer than RL and RND, on average, respectively. FARLEAD2 spends 3.25 seconds per step, on average,

due to real-time execution. Hence, in the case of regression testing, FARLEAD2 would replay an RL+GER witness 0.94 and 3.35 seconds slower than RL and RND, respectively. Since RL+GER achieves significantly higher effectiveness, these slightly slower witness replay times may be acceptable for the developer/tester.

7.3.3.5. RQ5: Levels of Information. Table 7.1 shows that every witness generator is 100 percent effective in every L4-STS. Performances are also similar, except for test scenario 014. RL+GER outperforms RND and RL in this test scenario. Still, every witness generator produces a witness for test scenario 014 in a reasonable time (around or under a minute), so all witness generators are preferable.

Under an L3-STS, RL+GER is 98.2 percent effective, 18.2 and 40.6 percent more than RL and RND, respectively. Similarly, RL+GER is consistently the most effective witness generator under all information levels. Again, except for L4-STSs, RL+GER outperforms RND and RL under every information level. Overall, RL+GER is more effective and performs better than RND and RL, regardless of the level of information.

7.3.3.6. RQ6: Test Scenario Complexity. Witness length measures of any L4 column in Table 2 reflect the complexities of our experimental test scenarios, with the notable exception of test scenario 015. Our manually created witness for test scenario 015 is eight steps long. However, experiments show that a six steps long witness exists for this test scenario. Specifically, this test scenario first creates a text note and then deletes it. Our manually generated test opens the text note, clicks the delete button, confirms the deletion with another click, and returns to the list to verify that the text note is gone. The automatically generated witness performs a long click on the text note and clicks the appearing trash icon, visibly removing it from the list. Hence, the true complexity of test scenario 015 is not higher than 014 but equal to 013. Otherwise, all the other test scenarios are in ascending order of complexity.

The rightmost three columns of Table 2 show that, for test scenarios 002-009, where the test complexity is four or less, all the witness generators have 80 percent or higher effectiveness. RND never gets back to 80 percent effectiveness at test scenarios 010-017. RL gets mixed results with 100 percent effectiveness for test scenarios 010, 011, and 015 but drops to 35 percent for test scenario 014. On the other hand, RL+GER gets 100 percent effectiveness for the same test scenarios but never drops below 67.5 percent. These results show that (1) increasing test complexity causes effectiveness problems, and (2) RL+GER is more robust to increasing test complexity than RND and RL.

In non-complex test scenarios 002-009, RL-GER outperforms RND and RL. In complex test scenarios 010-013 and 015, RL+GER keeps having better performance than RND and RL. However, for test scenarios 014, 016, and 017, RND outperforms RL and RL+GER consistently with around 1062-1082 seconds against 1786-2626 and 1395-3048 seconds, respectively. These results show that although RL and RL+GER fail fewer times than RND, the cost of failure is higher for RL and RL+GER.

7.3.3.7. Summary. Our experimental results show that, on average, RL+GER is more effective, generates witnesses for more test scenarios, and produces more witnesses within a constant testing budget than RL and RND. RL+GER is more effective than RL and RND, regardless of the level of information and the complexity of the test scenario.

The downsides of RL+GER are, (i) it may terminate later than RND for complex test scenarios, (ii) it may rarely produce no witness due to maximization bias caused by previous experience, and (iii) it produces longer witnesses on average, slightly increasing witness replay times. We argue that RL+GER's benefits outweigh its downsides because (i) it outperforms RND on average, (ii) false-lead experience maximization bias does not harm its overall effectiveness, and (iii) it covers more test scenarios, compensating for longer witnesses.

8. DISCUSSION

All the four test generators in this thesis, QBE, TCM, and FARLEAD/FARLEAD2, suffer from the inability of detecting the infeasibility of achieving their test adequacy criteria. We cannot say if an activity is unreachable, a bug does not exist, or a functional behavior is not implemented using QBE, TCM, and FARLEAD/FARLEAD2, respectively. However, we argue that an AUT for which TCM cannot find a crash is more reliable than an AUT for which TCM finds crashes. Also, if QBE cannot reach an activity or FARLEAD/FARLEAD2 cannot trigger a functional behavior, it's reasonable to assume that reaching the activity or triggering the functional behavior would be difficult for the user, too, which may encourage the developer to re-design an easier way for the users.

In theory, QBE, TCM, and FARLEAD/FARLEAD2 are challenging to generalize to all GUI applications on all platforms because

(i) Some AUTs such as messaging applications must be installed on multiple devices. Our test generators control only one device at a time, so they won't be able to test some critical functions of those AUTs.

(ii) GUI states between every two GUI action must be stable, i.e., not changing. However, in practice, system events and remote calls may also affect GUI output without any GUI action, confusing the test generator.

(iii) There must be a small, finite set of enabled actions at every GUI state. Some GUIs, especially games may have infinite or almost infinite set of enabled actions, leading to action explosion. If the developer does not mitigate this explosion via input-space partitioning, in theory, RL would be rendered no better than any random test generator.

(iv) The GUI must report crash logs, the activity name, and information on the GUI widgets so we can label the GUI states with Boolean propositions. However, this is not always the case in practice. For example, iOS devices provide different logs

and widget information than Android, and do not provide any activity names at all. Hence, even for the same AUT built for different devices (e.g. iOS and Android) may not behave the same for the test generator, leading an inevitable variance in effectiveness and performance depending on the GUI environment.

If a GUI environment is configured to provide stable GUI states, the input-space is partitioned enough to allow only small, finite sets of enabled actions at every GUI state, and the environment reports crashes, activities and its widgets, then our methods are applicable to that GUI environment in addition to the Android OS.

If a test scenario involves too many steps, it quickly becomes intractable for the test generator to generate a witness. The developer/tester should divide such test scenarios into successive test scenarios, for which the test generator generates tests in reasonable time. Optimizations such as ER also benefit from this divide and conquer strategy.

Even though we delegate the task of finding the low-level GUI actions to the test generator, we cannot guarantee it can generate a desired test on every Android device just because it generates that test on an Android device. Android devices come with different resolutions, memory, data, and OS versions. All these factors inevitably affect test generation. The developer/tester should execute the test generator on different devices to establish the AUTs reliability over a representative sample of devices. Note that the generated test is often not portable from a device to another without a smart self-healing technique adapting the test for the new device.

9. CONCLUSION

Summary of our contributions in this thesis are as follows.

(i) We describe the related work on Automated GUI Testing, especially for Android, and its shortcomings.

(ii) We improve the state-of-the-art in terms of automatic exploration of GUI screens and detecting unique crashes, with QBE and TCM.

(iii) We enable automated functional testing of GUI application through test scenarios with FARLEAD/FARLEAD2.

(iv) We use RL, a semi-supervised ML technique, in every test generator, improving both structural and functional testing.

(v) We improve RL effectiveness and performance with GER, allowing the exploitation of experience gathered during previous test generation tasks on new tasks.

(vi) We design a new test scenario language, STS, to make test scenarios both human-readable and unambiguously run-time monitorable.

(vii) We describe or test generators with overall figures, examples, and algorithms.

(viii) Our experimental results show that within a fixed testing time,
   (a) QBE covers more screens (activities) than other test generators,
   (b) Executing TCM on top of QBE detects the highest number of unique crashes, and
   (c) FARLEAD2 with GER provides the highest performance and effectiveness in verifying functional behavior.

Note that this thesis is not just the first work that enables functional testing but it also makes functional testing more practical via STSs. With this development, the developer/tester no longer needs to know coding to implement tests for automation. Still, somebody has to write the test scenarios in STS, which takes time and manual effort. Any future work regarding this thesis must mainly focus on removing the remaining human intervention, completely.

As future work, we will

(i) Investigate NLP techniques that could convert true natural language descriptions in GitHub issues or software requirement documents to STSs.

(ii) Learn multiple Q-values from multiple reward functions in the same episode, making the RL-engine multi-objective. A multi-objective RL could learn many behaviors in one go, increasing test generator performance and effectiveness.

(iii) Investigate methods that either perform an initial random crawl on the AUT or a static analysis on its binary to automatically create test scenarios. Such a crawler/analyzer may work in combination with the NLP transformers converting descriptions to test scenarios.

REFERENCES

1. Bloomberg, "Knight Shows How to Lose $440 Million in 30 Minutes", https://www.bloomberg.com/news/articles/2012-08-02/knight-shows-how-to-lose-440-million-in-30-minutes, accessed on September 1, 2018.

2. Wikipedia, "Morris Worm", https://en.wikipedia.org/wiki/Morris_worm, accessed on September 1, 2018.

3. Wikipedia, "Pentium FDIV Bug", https://en.wikipedia.org/wiki/Pentium_FDIV_bug, accessed on September 1, 2018.

4. National Institute of Standards and Technology (NIST), "The Economic Impacts of Inadequate Infrastructure for Software Testing", http://www.nist.gov/director/planning/upload/report02-3.pdf, accessed on July 29, 2016.

5. Venolia, G. D., R. DeLine and T. LaToza, "Software Development at Microsoft Observed", http://research.microsoft.com/apps/pubs/default.aspx?id=70227, accessed on July 29, 2016.

6. Burnim, J. and K. Sen, "Heuristics for Scalable Dynamic Test Generation", *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, 2008.

7. Boyer, R. S., B. Elspas and K. N. Levitt, "SELECT, a Formal System for Testing and Debugging Programs by Symbolic Execution", *Proceedings of the International Conference on Reliable Software*, 1975.

8. Piejko, P., "15 Mobile Web Predictions for 2020", https://deviceatlas.com/blog/15-mobile-web-predictions-2020, accessed on July 12, 2022.

9. Ceci, L., "Number of Monthly Google Play App Releases Worldwide 2019-2021",

https://www.statista.com/statistics/1020956/android-app-releases-worldwide/, accessed on December 3, 2021.

10. Rodríguez-Valdés, O., T. E. Vos, P. Aho and B. Marín, "30 Years of Automated GUI Testing: A Bibliometric Analysis", *International Conference on the Quality of Information and Communications Technology*, pp. 473–488, Springer, 2021.

11. Khan, M. E., F. Khan *et al.*, "A Comparative Study of White Box, Black Box and Grey Box Testing Techniques", *Int. J. Adv. Comput. Sci. Appl*, Vol. 3, No. 6, 2012.

12. Gultnieks, C., "F-Droid Benchmarks", https://f-droid.org/, accessed on July 12, 2022.

13. Hao, S., B. Liu, S. Nath, W. G. Halfond and R. Govindan, "PUMA: Programmable UI-automation for Large-scale Dynamic Analysis of Mobile Apps", *12th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pp. 204–217, 2014.

14. Azim, T. and I. Neamtiu, "Targeted and Depth-first Exploration for Systematic Testing of Android Apps", *ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pp. 641–660, 2013.

15. Choi, W., G. Necula and K. Sen, "Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning", *ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pp. 623–640, 2013.

16. Mao, K., M. Harman and Y. Jia, "Sapienz: Multi-Objective Automated Testing for Android Applications", *25th International Symposium on Software Testing and Analysis (ISSTA)*, pp. 94–105, 2016.

17. Moran, K., M. L. Vásquez, C. Bernal-Cárdenas, C. Vendome and D. Poshyvanyk, "Automatically Discovering, Reporting and Reproducing Android Application Crashes", *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pp. 33–44, 2016.

18. Machiry, A., R. Tahiliani and M. Naik, "Dynodroid: An Input Generation System for Android Apps", *9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2013.

19. Koroglu, Y., A. Sen, O. Muslu, Y. Mete, C. Ulker, T. Tanriverdi and Y. Donmez, "QBE: QLearning-Based Exploration of Android Applications", *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2018.

20. Azim, T., I. Neamtiu and L. M. Marvel, "Towards Self-healing Smartphone Software via Automated Patching", *29th ACM/IEEE International Conference on Automated Software Engineering (ASE)*, pp. 623–628, 2014.

21. Koroglu, Y. and A. Sen, "TCM: Test Case Mutation to Improve Crash Detection in Android", *Fundamental Approaches to Software Engineering*, 2018.

22. Koroglu, Y. and A. Sen, "Functional Test Generation from UI Test Scenarios Using Reinforcement Learning for Android Applications", *Software Testing, Verification and Reliability*, Vol. 31, No. 3, p. e1752, 2021.

23. Koroglu, Y., A. Sen and A. Akin, "Automated Functional Test Generation Practice for a Large-Scale Android Application", *Turkish National Software Engineering Symposium (UYMS)*, pp. 1–3, IEEE, 2020.

24. Laud, A. D., *Theory and Application of Reward Shaping in Reinforcement Learning*, Tech. rep., University of Illinois at Urbana-Champaign, 2004.

25. Google Developers, "Android UI/Application Exerciser Monkey", `http://developer.android.com/tools/help/monkey.html`, accessed on July 12, 2022.

26. Anand, S., M. Naik, M. J. Harrold and H. Yang, "Automated Concolic Testing of Smartphone Apps", *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE)*, 2012.

27. Yang, W., M. R. Prasad and T. Xie, "A Grey-box Approach for Automated GUI-model Generation of Mobile Applications", *16th International Conference on Fundamental Approaches to Software Engineering (FASE)*, pp. 250–265, 2013.

28. Mahmood, R., N. Mirzaei and S. Malek, "EvoDroid: Segmented Evolutionary Testing of Android Apps", *22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pp. 599–609, 2014.

29. Liu, Y., C. Xu, S.-C. Cheung and J. Lü, "Greendroid: Automated Diagnosis of Energy Inefficiency for Smartphone Applications", *IEEE Transactions on Software Engineering*, Vol. 40, No. 9, pp. 911–940, 2014.

30. Amalfitano, D., A. R. Fasolino, P. Tramontana, B. D. Ta and A. M. Memon, "MobiGUITAR: Automated Model-Based Testing of Mobile Apps", *IEEE Software*, Vol. 32, No. 5, pp. 53–59, 2015.

31. Zaeem, R. N., M. R. Prasad and S. Khurshid, "Automated Generation of Oracles for Testing User-Interaction Features of Mobile Apps", *IEEE International Conference on Software Testing, Verification, and Validation (ICST)*, 2014.

32. Linares-Vásquez, M., M. White, C. Bernal-Cárdenas, K. Moran and D. Poshyvanyk, "Mining Android App Usages for Generating Actionable GUI-based Execution Scenarios", *12th Working Conference on Mining Software Repositories (MSR)*, pp. 111–122, 2015.

33. Mirzaei, N., J. Garcia, H. Bagheri, A. Sadeghi and S. Malek, "Reducing Combinatorics in GUI Testing of Android Applications", *IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 559–570, IEEE, 2016.

34. Li, Y., Z. Yang, Y. Guo and X. Chen, "DroidBot: A Lightweight UI-Guided Test Input Generator for Android", *IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 2017.

35. Su, T., G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu and Z. Su, "Guided, Stochastic Model-based GUI Testing of Android Apps", *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017.

36. Cao, Y., G. Wu, W. Chen and J. Wei, "CrawlDroid: Effective Model-based GUI Testing of Android Apps", *Tenth Asia-Pacific Symposium on Internetware*, 2018.

37. Eler, M. M., J. M. Rojas, Y. Ge and G. Fraser, "Automated Accessibility Testing of Mobile Apps", *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, 2018.

38. Choi, W., "SwiftHand2: Android GUI Testing Framework", `https://github.com/wtchoi/swifthand2`, accessed on July 12, 2022.

39. Yan, J., L. Pan, Y. Li, J. Yan and J. Zhang, "LAND: A User-friendly and Customizable Test Generation Tool for Android Apps", *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2018.

40. Cao, C., J. Deng, P. Yu, Z. Duan and X. Ma, "Paraaim: Testing Android Applications Parallel at Activity Granularity", *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1, pp. 81–90, IEEE, 2019.

41. Su, T., Y. Yan, J. Wang, J. Sun, Y. Xiong, G. Pu, K. Wang and Z. Su, "Fully Automated Functional Fuzzing of Android Apps for Detecting Non-Crashing Logic Bugs", *Proceedings of the ACM on Programming Languages*, Vol. 5, No. OOPSLA, pp. 1–31, 2021.

42. Mariani, L., M. Pezze, O. Riganelli and M. Santoro, "AutoBlackTest: Automatic

Black-Box Testing of Interactive Applications", *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pp. 81–90, 2012.

43. Carino, S. and J. H. Andrews, "Dynamically Testing GUIs Using Ant Colony Optimization (T)", *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 138–148, 2015.

44. Esparcia-Alcázar, A. I., F. Almenar, M. Martínez, U. Rueda and T. Vos, "Q-learning Strategies for Action Selection in the TESTAR Automated Testing Tool", *6th International Conferenrence on Metaheuristics and nature inspired computing (META 2016)*, pp. 130–137, 2016.

45. Adamo, D., M. K. Khan, S. Koppula and R. Bryce, "Reinforcement Learning for Android GUI Testing", *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, pp. 2–8, 2018.

46. Vuong, T. A. T. and S. Takada, "A Reinforcement Learning Based Approach to Automated Testing of Android Applications", *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, p. 31–37, 2018.

47. Gomez, L., I. Neamtiu, T. Azim and T. Millstein, "RERAN: Timing- and Touch-sensitive Record and Replay for Android", *International Conference on Software Engineering (ICSE)*, 2013.

48. Hu, Y. and I. Neamtiu, "VALERA: An Effective and Efficient Record-and-replay Tool for Android", *International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 2016.

49. Fazzini, M., E. N. D. A. Freitas, S. R. Choudhary and A. Orso, "Barista: A Technique for Recording, Encoding, and Running Platform Independent Android

Tests", *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2017.

50. Falcone, Y., S. Currea and M. Jaber, "Runtime Verification and Enforcement for Android Applications with RV-Droid", *International Conference on Runtime Verification*, pp. 88–95, Springer, 2012.

51. Daian, P., Y. Falcone, P. O. Meredith, T. Serbanuta, S. Shiraishi, A. Iwai and G. Rosu, "RV-Android: Efficient Parametric Android Runtime Verification, a Brief Tutorial", *Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings*, Vol. 9333 of *Lecture Notes in Computer Science*, pp. 342–357, Springer, September 2015.

52. Sun, H., A. Rosà, O. Javed and W. Binder, "ADRENALIN-RV: Android Runtime Verification Using Load-Time Weaving", *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2017.

53. Zhang, P., K. Cheng and J. Gao, "Android-SRV: Scenario-Based Runtime Verification of Android Applications", *International Journal of Software Engineering and Knowledge Engineering*, Vol. 28, No. 02, pp. 239–257, 2018.

54. Hasanbeig, M., A. Abate and D. Kroening, "Certified Reinforcement Learning with Logic Guidance", *arXiv preprint arXiv:1902.00778*, 2019.

55. Hasanbeig, M., A. Abate and D. Kroening, "Logically-Constrained Neural Fitted Q-Iteration", *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*, pp. 2012–2014, International Foundation for Autonomous Agents and Multiagent Systems, 2019.

56. Toro Icarte, R., T. Q. Klassen, R. Valenzano and S. A. McIlraith, "Teaching Multiple Tasks to an RL Agent Using LTL", *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '18, pp.

452–461, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 2018.

57. Wen, M., R. Ehlers and U. Topcu, "Correct-by-synthesis Reinforcement Learning with Temporal Logic Constraints", *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2015.

58. Baek, Y.-M. and D.-H. Bae, "Automated Model-based Android GUI Testing Using Multi-level GUI Comparison Criteria", *31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 238–249, 2016.

59. Mao, H., M. Alizadeh, I. Menache and S. Kandula, "Resource Management with Deep Reinforcement Learning", *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pp. 50–56, ACM, 2016.

60. Arel, I., C. Liu, T. Urbanik and A. Kohls, "Reinforcement Learning-based Multi-agent System for Network Traffic Signal Control", *IET Intelligent Transport Systems*, Vol. 4, No. 2, pp. 128–135, 2010.

61. Silver, D., T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan and D. Hassabis, "Mastering Chess and Shogi by Self-play with a General Reinforcement Learning Algorithm", *arXiv preprint arXiv:1712.01815*, 2017.

62. Mnih, V., K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning", *arXiv preprint arXiv:1312.5602*, 2013.

63. Zhou, Z., X. Li and R. N. Zare, "Optimizing Chemical Reactions with Deep Reinforcement Learning", *ACS central science*, Vol. 3, No. 12, pp. 1337–1344, 2017.

64. Sutton, R. S. and A. G. Barto, *Reinforcement Learning: An Introduction*, The MIT Press, 2nd edn., 2018.

65. Drugan, M., "Reinforcement Learning versus Evolutionary Computation: A Survey on Hybrid Algorithms", *Swarm and Evolutionary Computation*, 03 2018.

66. Lin, L.-J., "Self-improving Reactive Agents Based on Reinforcement Learning, Planning and Teaching", *Machine Learning*, Vol. 8, No. 3-4, pp. 293–321, 1992.

67. Watkins, C. J. C. H., *Learning from Delayed Rewards*, Ph.D. Thesis, King's College, Cambridge, 1989.

68. Google Developers, "The Android Emulator", `https://developer.android.com/studio/run/emulator.html`, accessed on July 12, 2022.

69. Qin, F., Z. Zheng, X. Li, Y. Qiao and K. S. Trivedi, "An Empirical Investigation of Fault Triggers in Android Operating System", *IEEE 22nd Pacific Rim International Symposium on Dependable Computing (PRDC)*, pp. 135–144, 2017.

70. Zeller, A., "Yesterday, My Program Worked. Today, It Does Not. Why?", *7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-7)*, pp. 253–267, 1999.

71. Bolton, D., "88 Percent of People Will Abandon an App Because of Bugs", `https://www.applause.com/blog/app-abandonment-bug-testing`, accessed on July 2, 2017.

72. Barr, E. T., M. Harman, P. McMinn, M. Shahbaz and S. Yoo, "The Oracle Problem in Software Testing: A Survey", *IEEE transactions on software engineering*, Vol. 41, No. 5, pp. 507–525, 2014.

73. Koroglu, Y., "Black is Missing from the Color Palette (Issue #75)", `https://github.com/SecUSo/privacy-friendly-notes/issues/75`, accessed on July 12, 2022.

APPENDIX A: QBE's Experimental AUT Characteristics

Table A.1. AUT Characteristics.

	Training Set			**Test Set**			**F-Droid**		
Characteristics	Min	Max	Mean	Min	Max	Mean	Min	Max	Mean
Size (MB)	0.01	14.51	1.29	0.02	17.48	1.8	0.01	157.2	2.29
KInstructions	0.01	491.9	101.7	0.9	522.2	74.17	0.01	1395	107.4
KMethods	0.01	49.9	11.4	0.18	38.15	6.51	0.01	157.7	11.3
# Activities	1	37	4.54	1	28	6.3	0	123	5.79
# Permissions	0	31	8.44	0	24	6.4	0	168	8.4

Table A.1 shows the characteristics of our training and tests sets in more detail. We argue that our training set has similar charasteristics to our test set in terms of size, number of instructions, methods, activities, and permissions.

APPENDIX B: QBE's Experimental Coverage Results

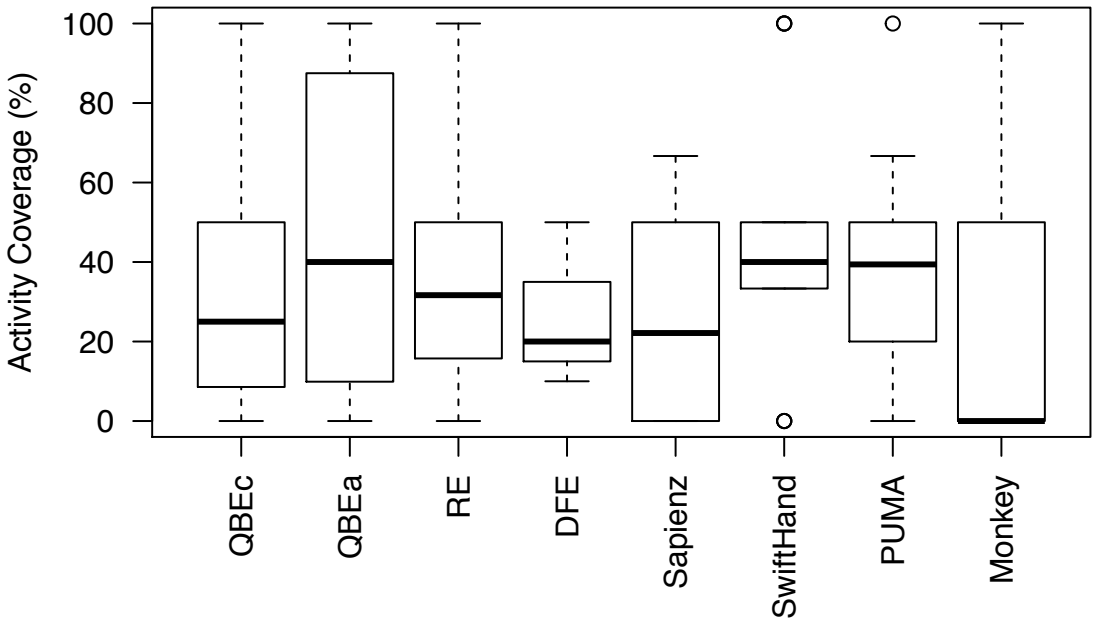


Figure B.1. Boxplots of Activity Coverages for Three Runs by Tool.

Figure B.1 gives more insight on QBE's improvement on activity coverage. From this figure, it is trivial to see when trained for crashes, QBEc performs similar to Random Exploration (RE). Hence, we deduce locating crashes and covering new activities yield incompatible general patterns accross Android applications. However, when trained for coverage, QBEa can achieve more than 80% coverage on some applications, which outperforms any other test generator.

APPENDIX C: FARLEAD's Experimental LTL Formulae

The LTL formulae related to the experimental test scenarios of FARLEAD:

ϕ_A:
(a) $\bigcirc([\text{activity} \sim \text{Main}]\,\mathcal{U}([\text{activity} \sim \text{About}] \wedge \bigcirc([\text{activity} \sim \text{About}]\,\mathcal{U}[\text{activity} \sim \text{Main}]))$
(b) $\bigcirc(([\text{activity} \sim \text{Main}] \wedge \text{actionType} = \text{click})\,\mathcal{U}([\text{activity} \sim \text{About}] \wedge \bigcirc([\text{actionType} = \text{back}]\,\mathcal{U}[\text{activity} \sim \text{Main}]))$
(c) $\bigcirc((([\text{actionType} = \text{click}] \wedge [\text{actionDetail} \sim \text{About}]) \wedge [\text{activity} \sim \text{About}]) \wedge \bigcirc([\text{actionType} = \text{back}]\,\mathcal{U}\text{activity} \sim \text{Main}))$

ϕ_B:
(a) $\bigcirc([\text{activity} \sim \text{Main}]\,\mathcal{U}([\text{activity} \sim \text{Settings}] \wedge \bigcirc([\text{activity} \sim \text{Settings}]\,\mathcal{U}[\text{activity} \sim \text{Main}]))$
(b) $\bigcirc(([\text{activity} \sim \text{Main}] \wedge \text{actionType} = \text{click})\,\mathcal{U}([\text{activity} \sim \text{Settings}] \wedge \bigcirc([\text{actionType} = \text{back}]\,\mathcal{U}[\text{activity} \sim \text{Main}]))$
(c) $\bigcirc((([\text{actionType} = \text{click}] \wedge [\text{actionDetail} \sim \text{About}]) \wedge [\text{activity} \sim \text{About}]) \wedge \bigcirc([\text{actionType} = \text{back}]\,\mathcal{U}\text{activity} \sim \text{Main}))$

ϕ_C:
(a) N/A
(b) $\bigcirc([\text{actionType} = \text{pauseresume}] \wedge [\text{activity} \sim \text{Main}])$
(c) $\bigcirc([\text{actionType} = \text{pauseresume}] \wedge [\text{activity} \sim \text{Main}])$

ϕ_D:
(a) $\top\,\mathcal{U}[\text{screen} = \text{off}]$
(b) $\bigcirc([\text{actionType} = \text{idle}] \wedge [\text{screen} = \text{off}])$
(c) $\bigcirc([\text{actionType} = \text{idle}] \wedge [\text{screen} = \text{off}])$

ϕ_E:
(a) $\bigcirc([\text{activity} \sim \text{Main}]\,\mathcal{U}(([\text{activity} \sim \text{Settings}] \wedge [\text{checked} = \text{true}]) \wedge ((([\text{checked} = \text{true}] \wedge [\text{activity} \sim \text{Settings}])\,\mathcal{U}([\text{checked} = \text{false}] \wedge \bigcirc([\text{activity} \sim \text{Main}] \wedge \bigcirc([\text{activity} \sim \text{Settings}] \wedge [\text{checked} = \text{false}])))))$
(b) $\bigcirc(([\text{actionType} = \text{click}] \wedge [\text{activity} \sim \text{Main}])\,\mathcal{U}(([\text{activity} \sim \text{Settings}] \wedge [\text{checked} = \text{true}]) \wedge \bigcirc((([\text{actionType} = \text{click}] \wedge [\text{activity} \sim \text{Settings}]) \wedge [\text{checked} = \text{true}])\,\mathcal{U}([\text{checked} = \text{false}] \wedge \bigcirc(([\text{actionType} = \text{back}] \wedge [\text{activity} \sim \text{Main}]) \wedge \bigcirc(([\text{actionType} = \text{click}] \wedge [\text{activity} \sim \text{Main}])\,\mathcal{U}([\text{activity} \sim \text{Settings}] \wedge [\text{checked} = \text{false}])))))))$
(c) $\bigcirc(([\text{actionDetail} \sim \text{Settings}] \wedge [\text{checked} = \text{true}]) \wedge \bigcirc((([\text{actionType} = \text{click}] \wedge [\text{actionObjectID} = 0{:}0{:}0{:}0{:}0]) \wedge [\text{checked} = \text{false}]) \wedge \bigcirc([\text{actionType} = \text{back}] \wedge \bigcirc([\text{actionDetail} \sim \text{Settings}] \wedge [\text{checked} = \text{false}]))))$

ϕ_F:
(a) $\bigcirc([\text{activity} \sim \text{New}] \wedge \bigcirc([\text{activity} \sim \text{Offline}] \wedge \bigcirc[\text{text} \sim \text{moved}]))$
(b) $\bigcirc(([\text{actionType} = \text{click}] \wedge [\text{activity} \sim \text{New}]) \wedge \bigcirc(([\text{actionType} = \text{click}] \wedge [\text{activity} \sim \text{Offline}]) \wedge \bigcirc([\text{actionType} = \text{chessmove}] \wedge [\text{text} \sim \text{moved}])))$
(c) $\bigcirc([\text{actionDetail} \sim \text{Offline}] \wedge \bigcirc([\text{actionDetail} \sim \text{Play}] \wedge \bigcirc([\text{actionType} = \text{chessmove}] \wedge [\text{text} \sim \text{moved}])))$

ϕ_G:
(a) $\bigcirc([\text{activity} \sim \text{Main}]\,\mathcal{U}([\text{activity} \sim \text{New}] \wedge \bigcirc([\text{activity} \sim \text{New}]\,\mathcal{U}([\text{activity} \sim \text{Offline}] \wedge \bigcirc([\text{activity} \sim \text{Offline}]\,\mathcal{U}([\text{text} \sim \text{moved}] \wedge \bigcirc([\text{activity} \sim \text{Offline}]\,\mathcal{U}([\text{activity} \sim \text{New}] \wedge \bigcirc([\text{activity} \sim \text{New}]\,\mathcal{U}([\text{activity} \sim \text{Offline}] \wedge [\text{text} \sim \text{moved}])))))))))$
(b) $\bigcirc(([\text{activity} \sim \text{Main}] \wedge [\text{actionType} = \text{click}])\,U([\text{activity} \sim \text{New}] \wedge \bigcirc(([\text{actionType} = \text{click}] \wedge [\text{activity} \sim \text{Offline}]) \wedge \bigcirc(([\text{actionType} = \text{chessmove}] \wedge [\text{text} \sim \text{moved}]) \wedge \bigcirc([\text{actionType} = \text{click}]U([\text{activity} \sim \text{New}] \wedge \bigcirc(([\text{actionType} = \text{click}] \wedge [\text{activity} \sim \text{Offline}]) \wedge [\text{text} \sim \text{moved}])))))))$
(c) $\bigcirc([\text{actionDetail} \sim \text{Offline}] \wedge \bigcirc([\text{actionDetail} \sim \text{Play}] \wedge \bigcirc([\text{actionType} = \text{chessmove}] \wedge \bigcirc([\text{actionDetail} \sim \text{New}] \wedge \bigcirc([\text{actionDetail} \sim \text{Play}] \wedge [\text{text} \sim \text{moved}])))))$

ϕ_H:
(a) $\bigcirc([\text{activity} \sim \text{Main}]\,\mathcal{U}([\text{activity} \sim \text{New}] \wedge \bigcirc([\text{activity} \sim \text{New}]\,\mathcal{U}([\text{activity} \sim \text{Offline}] \wedge \bigcirc([\text{activity} \sim \text{Offline}]\,\mathcal{U}([\text{text} \sim \text{moved}] \wedge \bigcirc([\text{activity} \sim \text{Offline}]\,\mathcal{U}([\text{activity} \sim \text{New}] \wedge \bigcirc([\text{activity} \sim \text{New}]\,\mathcal{U}([\text{activity} \sim \text{Offline}] \wedge [\text{text} \sim \text{moved}])))))))))$
(b) $\bigcirc(([\text{text} \sim \text{OK}] \wedge [\text{actionType} = \text{click}])\,\mathcal{U}([\text{activity} \sim \text{Main}] \wedge \bigcirc(([\text{activity} \sim \text{Main}] \wedge [\text{actionType} = \text{click}])\,\mathcal{U}([\text{text} \sim \text{sketch}] \wedge \bigcirc(([\text{activity} \sim \text{Main}] \wedge [\text{actionType} = \text{click}])\,\mathcal{U}([\text{activity} \sim \text{Sketch}] \wedge \bigcirc(([\text{activity} \sim \text{Sketch}] \wedge [\text{actionType} = \text{click}])\,\mathcal{U}([\text{objectID} \sim 18] \wedge \neg[\text{objectID} \sim 19])))))))$
(c) $\bigcirc(([\text{actionType} = \text{click}] \wedge [\text{actionDetail} \sim \text{OK}]) \wedge \bigcirc(([\text{actionType} = \text{click}] \wedge [\text{actionDetail} \sim +]) \wedge \bigcirc(([\text{actionType} = \text{click}] \wedge [\text{actionDetail} \sim \text{sketch}]) \wedge \bigcirc(([\text{actionType} = \text{click}] \wedge [\text{actionDetail} \sim \text{colorSelector}])\,\mathcal{U}([\text{objectID} \sim 18] \wedge \neg[\text{objectID} \sim 19]))))$

ϕ_I:
(a) N/A
(b) $\bigcirc(([\text{activity} \sim \text{Main}] \wedge [\text{actionType} = \text{click}])\,\mathcal{U}([\text{text} \sim \text{New text}] \wedge \bigcirc(([\text{actionType} = \text{click}] \wedge [\text{activity} \sim \text{TextNote}]) \wedge \bigcirc([\text{actionType} = \text{back}]\,\mathcal{U}[\text{text} \sim \text{Note 1}])))$
(c) $\bigcirc(([\text{actionType} = \text{click}] \wedge [\text{actionDetail} \sim \text{OK}]) \wedge \bigcirc(([\text{actionType} = \text{click}] \wedge [\text{actionDetail} \sim +]) \wedge \bigcirc(([\text{actionType} = \text{click}] \wedge [\text{actionDetail} \sim \text{text}]) \wedge \bigcirc([\text{actionType} = \text{back}]\,\mathcal{U}[\text{text} \sim \text{Note 1}]))))$

APPENDIX D: On the Figures of This Thesis

Figure 3.1 is available under the Creative Commons CC0 1.0 Universal Public Domain Dedication at Wikimedia Commons, accessed on 20th of June, 2022. This thesis respects the constraints the Creative Commons CC0 1.0 Universal Public Domain Dedication license with respect to this figure.

All the other figures are the author's own work, published in journals and conferences. This thesis respects all the publishers' rights with respect to these figures, obeying their reuse policies.