Università degli Studi di Napoli "Federico II"

Scuola Politecnica e delle Scienze di Base

Dottorato di Ricerca in Ingegneria Informatica ed Automatica

Ciclo XXVIII

May 2016

Tesi di Dottorato

Automated GUI Testing Techniques for Android Applications

Tutori:
Prof. Anna Rita Fasolino
Prof. Porfirio Tramontana

Candidato: Amatucci Nicola

Abstract

Mobile devices are integral parts of our daily lives; a little computer in our pocket has became a faithful assistant both for work than for amusement. The availability of mobile applications (commonly referred as *apps*) has made more and more useful bringing these devices with us everyday. The number of such applications in these years has faced a tremendous growth due to the market attractiveness [106]; according to Forbes[1], by 2017 more than 270 billion mobile applications will be downloaded worldwide. The quality of a mobile application is a major concern for developers, users and application stores [32]. According to a survey conducted by SmartBear[2] from October to December 2013 nearly 50% of consumers will delete a mobile app if they encounter a bug. So, testing mobile applications to prevent the occurrence of software exceptions in production can be considered one of the key factor influencing its quality together with the market response. As today, in literature many techniques have been presented aiming at testing mobile applications. In particular, many of them have been presented in the context of GUI Testing.

The research activity described in this thesis is focused on proposing novel techniques and tools in the field of Automated GUI Testing for Mobile Applications. In particular, the work is targeted to the Android Operating System, that currently is the dominating operating system in the mobile devices market [54], although the results can be generalized to other mobile platforms.

[1]http://blogs-images.forbes.com/niallmccarthy/files/2014/10/Giant-social-apps_Forbes.jpg
[2]https://smartbear.com/news/news-releases/the-state-of-mobile-testing-2014/

Acknowledgements

Non sono bravo a scrivere i ringraziamenti, sono più propenso a farli di persona, ma ci provo cercando di non lasciar fuori nessuno, perché, diciamo la verità, se non fosse stato per le tante persone che mi hanno accompagnato in questo viaggio, probabilmente non sarei qui adesso a scrivere queste parole.

Ringrazio i miei tutor, la professoressa Anna Rita Fasolino ed il professor Porfirio Tramontana, che, insieme all'ingegner Domenico Amalfitano, qualche anno fa mi hanno accordato un'enorme fiducia concedendomi il privilegio di far parte di questo gruppo di ricerca. Senza di loro non sarei qui e non sarei potuto crescere così tanto sia professionalmente che personalmente. Sono stati un punto di riferimento costante in questo percorso, mi hanno guidato ed aiutato e spero di non aver deluso troppo le loro aspettative e di aver dato loro qualche soddisfazione. Sono davvero delle persone eccezionali; se non fosse stato per loro non sarei qui e li ringrazio dal profondo del mio cuore.

Ringrazio tutti i ragazzi che in questi anni hanno frequentato il Laboratorio 4.04 e che lo frequentano ancora. Senza di loro, questo periodo non sarebbe stato così bello, allegro e produttivo. Ringrazio tutti quelli che ho conosciuto qui all'Università di Napoli, professori, ricercatori, dottorandi, collaboratori, personale amministrativo e non, tesisti, studenti e tutte le persone che in questi anni ho incontrato. Alcuni di loro li ringrazio particolarmente e, forse, non li ringrazierò mai abbastanza; questo pensiero è dedicato in particolar modo a loro e soprattutto a quelli che posso chiamare amici.

Voglio ringraziare i miei genitori e mia nonna che mi hanno incoraggiato, supportato, sostenuto e sopportato giorno per giorno e continuano a sostenermi. Grazie mille davvero per tutto quello che avete fatto in questi anni, da quando sono nato; senza di voi nulla sarebbe stato possibile.

Ringrazio i miei suoceri, i miei cognati e le mie cognate che mi hanno sempre

aiutato e sostenuto in ogni momento bello o brutto e ora fanno parte della mia famiglia.

Ringrazio tutti gli amici e parenti che, anche se forse in questi ultimi anni ho trascurato un po', sono sempre nei miei pensieri e nel mio cuore.

Infine, con tutto il mio cuore, voglio ringraziare mia moglie Liliana, che ho avuto la fortuna di sposare quasi un anno fa, e che in questi anni mi ha davvero donato tanto amore e che amo con tutto me stesso; anche nei momenti di difficoltà è stata sempre accanto a me e lo è ancora oggi, per fortuna. Aspettiamo un bambino o una bambina (ancora non lo sappiamo) e questo oltre a darmi immensa gioia, mi dà forza e coraggio di affrontare tutto quello che verrà. Grazie, infinitamente. Dedico a voi questa tesi, come dedico a voi la mia vita, ogni giorno.

Spero di non aver dimenticato nessuno in questi miei, seppur sintetici, ringraziamenti[3].

Grazie a tutti,
Nicola

[3]Ci tengo a precisare che tutte le persone citate in queste pagina hanno svolto un ruolo fondamentale nella stesura della tesi, ma che ogni errore o imprecisione è imputabile soltanto a me

A Liliana, la donna che amo,

al frutto del nostro amore.

Contents

List of Figures

List of Tables

Abbreviations

- **OS:** Operating System
- **UI:** User Interface
- **GUI:** Graphical User Interface
- **SDK:** Software Development Kit
- **NDK:** Native Development Kit
- **AUT:** Application Under Test
- **XML:** eXtensible Markup Language
- **URI:** Uniform Resource Identifier
- **IPC:** Inter Process Communication
- **SMS:** Short Message Service
- **AUT:** Application Under Test
- **IDE:** Integrated Development Environment
- **LOC:** Line of Code
- **EDS:** Event Driven Systems
- **MSC:** Message Sequence Charts
- **EFG:** Event Flow Graphs
- **UML:** Unified Modeling Language
- **USB:** Universal Serial Bus
- **CAAA:** Context-Aware Adaptive Application
- **FSM:** Finite State Machine
- **API:** Application Programming Interface

- **MFT:** Monkey Fuzz Testing
- **JVM:** Java Virtual Machine
- **ART:** Android RunTime
- **AIDL:** Android Interface Definition Language
- **ADT:** Android Developer Toolkit
- **AVD:** Android Virtual Device
- **FIFO**: First-In-First-Out
- **LIFO**: Last-In-First-Out

CHAPTER 1

Introduction

1.1 Introduction

Mobile Applications can be considered Event Driven Systems [8]; they are able to react both to events related to User Interactions both to system events [68], i.e. events generated by the device hardware platform or by the running environment. For testing a Mobile Application it is possible to extend and adapt testing techniques originally designed for Event Driven Systems. Mobile Applications are mainly based on a Graphical User Interface (GUI) front-end, whose behavior can further be context-sensitive [120], i.e. they can react to changes in the orientation of the device, in the user location, in the status of the battery and so on; given that, the System Testing activity can be performed by generating and executing sequences of events, that sample the input space of the application [89].

When testing mobile applications, their characteristics and the ones of the running environment should be taken into account. In fact, differently from applications deployed on platforms like desktop or client systems, mobile devices and frameworks can expose the applications to new kinds of bugs [51]. This challenge has been acknowledged by the researchers that have proposed many techniques and tools about Mobile Applications Testing. Some important contributions in the literature are related to testing automation.

According to Muccini et al. [85] completely automatic techniques supporting testing should represent a very important value in the development of mobile applications, since they can be carried out quickly and with saving of human resources. Unfortunately, Joorabchi et al. [56] on the basis of interviews to 12 senior mobile app developers concluded that manual testing is prevalent for mobile applications and that GUI testing is challenging to automate. A recent study of Kochhar et al. [60] based on 627 open source Android projects hosted on GitHub showed that practices related to testing automation are rarely diffused in Android testing. In particular, they found that only 14% of the apps they have analyzed contains executable test cases and only 4% of apps have test cases able to cover more than 40% of the source code of the applications.

In the mobile domain, Android is the most popular Operating System [54], with an increasing number of applications available in the Google Play Market; for this reason along with its open-source nature and the use of the Java programming language, many of the techniques presented in literature are implemented in the context of Android Platform.

1.2 Thesis Goal

In this thesis I will present some contributions that try to address the challenges related to the automation of event-based testing of Mobile Applications. In particular, due to the reasons already exposed in Section 1.1, I focused on the Android Operating System.

In a first part of this work I will present the details of the design and the implementation of a novel automated GUI testing tool where I have implemented a set of techniques presented in the literature and some original techniques. The other contributions presented in this thesis are related to the improvement of the performance of such techniques, both in terms of effectiveness and efficiency.

Regarding the effectiveness, I will present two contributions. Firstly I will show an implementation of the presented techniques intended to exercise the peculiar features of mobile applications related to non-user events. Furthermore, I will present an automated technique, based on genetic algorithms, that has been designed and implemented to be used in conjunction with the previously described techniques. Also, I will present two contributions that aim to increase the efficiency of the automated testing processes. In particular, I will describe a stopping criterion for evaluating when a Random testing technique achieves an optimal termination point, that represents a good compromise between effectiveness and efficiency. Another contribution of this thesis regards the proposal and the implementation of a technique for parallel and distributed execution of automatic testing process. For each contribution I will present the results of experimentation carried out on real world applications.

1.3 Thesis Outline

The dissertation is organized as follows.

Chapter 2 provides some backgrounds about the Android Operating System and the Android Software Development Kit; it also summarizes some related work about the topic of GUI Testing, focusing in particular on the ones related to Automated GUI Testing Techniques for Android Applications.

Chapter 3 presents a generic algorithm for automated GUI testing techniques and describes its implementation in the *AndroidRipper* tool.

Chapter 4 and Chapter 5 present two contributions aimed at improving the effectiveness of the techniques described in Chapter 3. In particular, Chapter 4 presents an extended version of the *AndroidRipper* tool able to exercise context-sensitive applications [120]. Chapter 5 describes *AGRippin* (that is an acronym for Android Genetic Ripping) a search based testing technique applicable to Android applications with the purpose to generate test suites that are both effective in terms of coverage of the source code and efficient in terms of number of generated test cases.

Chapter 6 and Chapter 7 present two contributions aimed at incrementing the efficiency of the techniques described in Chapter 3. In Chapter 6 is addressed the problem of stopping a random testing process at a cost-effective point. Chapter 7, instead, presents a parallel approach for automated GUI Testing of Android Applications.

This thesis includes materials from the following research papers, already published in peer-reviewed conferences and journals:

Domenico Amalfitano, **Nicola Amatucci**, Anna Rita Fasolino, Porfirio Tramontana. *Considering Context Events in Event-Based Testing of Mobile Applications.* In Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on, pages 126–133, March 2013

Domenico Amalfitano, **Nicola Amatucci**, Anna Rita Fasolino, Ugo Gentile, Gianluca Mele, Roberto Nardone, Valeria Vittorini, and Stefano Marrone. *Improving code coverage in android apps testing by exploiting patterns and automatic test case generation.* In Proceedings of the 2014 international workshop on Long-term industrial collaboration on software engineering (WISE 2014), 2014

Domenico Amalfitano, **Nicola Amatucci**, Anna Rita Fasolino, Porfirio Tramontana, Emily Kowalczyk, and Atif Memon. *Exploiting the saturation effect in automatic random testing of android applications.* In The Proceedings of the 2nd ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft 2015), 2015.

Domenico Amalfitano, **Nicola Amatucci**, Anna Rita Fasolino, Porfirio Tramontana. *AGRippin: a novel search based testing technique for Android applications.* In The Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile (DeMobile 2015), 2015.

Domenico Amalfitano, **Nicola Amatucci**, Anna Rita Fasolino, Porfirio Tramontana. *A conceptual framework for the comparison of fully automated gui testing techniques.* In The Proceedings of the Sixth International Workshop on Testing Techniques for Event BasED Software, 2015.

CHAPTER 2

Background & Related Work

In this chapter I will introduce some details of the Android Operating System and the Android SDK; then I will report some related work about Event-Based testing of Android applications, focusing in particular on Automated Testing Techniques for Android Applications with the purpose of assessing the importance and showing the state of the art of this research topic.

2.1 The Android Operating System

Android is an open source platform, designed for handset devices like mobile phones and tablets. The Android Open Source Project is maintained by the Open Handset Alliance, a group of hardware and electronics manufacturers, software companies and network operators, lead by Google. It is built on top of the Linux Kernel and is released under the Apache 2.0 license: everyone can download and modify Android, but official releases should be approved by Google. From the point of view of a manufacturer, this represent an enormous advantage, so more and more devices based on the Android Operating System become available every day. The increasing spread of such devices has lead during the years to an exponential growth of the number of applications available for the OS. Moreover, by using the Java language, the Android Studio IDE and the other free development tools, a developer can easily write and publish Android Applications.

In the following we are going to detail the Android Framework Architecture and focus on some development concept that will be useful for better understanding the testing techniques for such applications.

2.1.1 The Android Framework Architecture

The Android Framework has the layered architecture shown in Fig. 2.1.

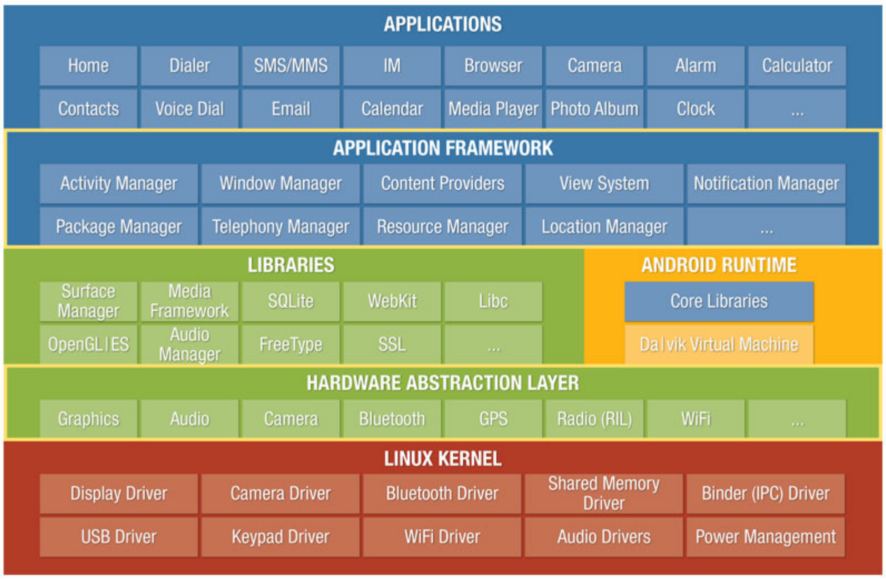


Figure 2.1: Android Framework Architecture

At the bottom of the stack there is the **Linux Kernel** layer, responsible of providing the core services of the system, i.e. security, memory management, process scheduling, networking services, device drivers and so on. This layer abstracts the hardware layer to the upper levels, allowing the hardware independence of the Android Framework.

On top of the kernel layer there is the **Hardware Abstraction Layer (HAL)**, that is the standard interface that allows Android applications to be agnostic about lower-level driver implementations.

The **Libraries** layer offers services that can be exploited by the upper levels, like multimedia management (Media Framework), font management (FreeType), data storage (SQLite), embeddable web browser (WebKit), 3D Rendering and so on. Within this layer there is the **Android Runtime**, featuring the Core Libraries and the Virtual Machine Implementation. Android version before Android Lollipop (5.0) were equipped with the Dalvik Virtual Machine (DVM); starting from Android Lollipop, the DVM has been replaced with Android RunTime (ART) as runtime environment.

Calls made by applications are handled by the **Application Framework** layer. This layer offers reusable components for building Android applications. As an example the level offers the following components:

- **View System**: it allows to easily build the GUI of an application.
- **Content Provider**: it eases the communication and the sharing of data between applications.
- **Resource Manager**: it manages the access to resources like strings, images, xml and so on.
- **Notification Manager**: it allows an application to send notification to the user.
- **Activity Manager**: it handles the life cycle of the applications running and manages what is rendered on the screen.
- **Location Manager**: it eases the retrieval of the current location of the device.
- **Package Manager**: it allows the management of the applications installed on the device.

The top layer is the **Applications** layer, which handles all applications that are installed on the device, some of which are shipped within the Android OS, like the email client, the telephony application, the contact manager an so on.

2.1.2 Core Concepts of the Android Platform

Android applications can be developed by using the Android Software Development Kit (SDK)[1] and can be made up by different types of components:

- **Activities** are components that display a Graphical User Interface (GUI) on the screen (typically described by a set of XML files) that mobile users can interact with. They are also responsible for monitoring and reacting to such interactions.
- **Services** are components that do not display any GUI Interface, that usually run in the background and perform long term tasks. Services can be started as independent task by calling the method *startService()* or through application bindings. A bound service is subjected to an application, its life cycle is bound to that of the application.
- **Content Providers** can be seen as databases for the applications. Data can be shared across applications using a standard interface through Content Providers. To access data, applications must have the needed permissions and the URI of the Content Provider. Android offers itself a set of Content Providers for Contacts, Messages and so on.
- **Broadcast Receivers** listen and handle events related to particular states of either the system or other applications, like when a new Message has been received or when the OS has finished its initialization.

[1]Android applications or libraries can be also developed using the Native Development Kit (NDK)

2.1.2.1 Inter Process Communication (IPC)

IPC mechanisms in Android[2] include:

- **Intents (along with Bundles)** that are the preferred mechanism for asynchronous IPC in Android. An Intent is a message consisting of the data together with the action to be performed. The data is encapsulated into a *Bundle*, that is a key-value data structure where a key is an instance of the *String* object and a value is an instance of an object implementing the *android.os.Parcelable* interface. The specific operation is univocally identified by a constant *String* value. Intents sent directly to a known recipient are called *Explicit Intents*; *Implicit Intents*, instead, are Intents sent in broadcast to all the registered receivers. Intent are commonly used to: start an Activity; start, stop and bind a Service; query a Content Provider.

- **Binders or Messengers (with a Service)** that are the preferred mechanisms for RPC-style IPC in Android. A Bound Service[3] is like a server which allows clients (as an example, Activities) to bind to the Service and then send requests and receive responses. If the Service runs in the same process as the client a Binder should be implemented; a Messager, instead, is needed when the communication is across different processes.

- **Broadcast Receivers** that are components of an application that are intended to receive Intents from other applications that own the needed permissions.

[2]http://developer.android.com/training/articles/security-tips.html#IPC
[3]http://developer.android.com/guide/components/bound-services.html

2.1.2.2 Android Activities

An Activity is the component of an Android Application that is intended to show on the screen an instance of its GUI and is able to react to User Interactions or System Events. Typically an application is made up of one or more Activities. Each Activity can be launched both by the User both by the System; the one that is launched when the application starts is called *Main Activity*. Only one Activity at a time can be on the screen; when a new Activity is created the one showing will be paused and moved by the OS in the *Back Stack*[4], while the new one is showed on the screen; when the new Activity is closed (because it terminates its execution by returning a result or because the user presses the BACK button on the device) it is removed by the back stack and the previous one is resumed and showed.

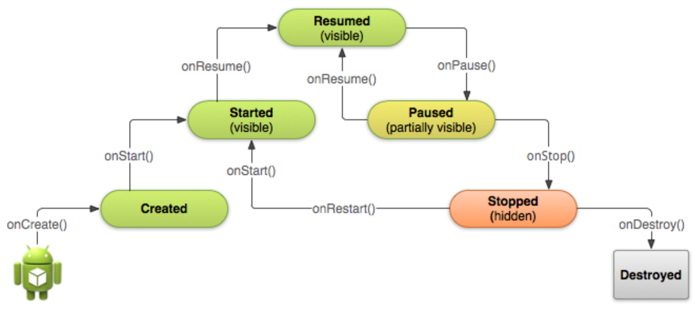


Figure 2.2: Activity States

The diagram in Figure 2.2 illustrates the possible states of an activity during its lifetime.

- **Created:** when an Activity is in the *Created* state, the needed resources have been allocated.

- **Started:** an Activity is considered *Started* if it is in the foreground, i.e. on

[4]http://developer.android.com/guide/components/tasks-and-back-stack.html

the top of the stack of the activities. This Activity is the only one showed to the user and that can react to user's interactions; it has the highest priority to allocate resources and can be killed by the OS only in some extreme situations that cause the UI to become unresponsive.

- **Paused:** when an Activity does not occupy user focus (as an example, when it is hidden by a new Activity or is partially visible or transparent) is considered *Paused*; in this state the Activity is still on the screen, owns an high priority to allocate resources and is attached to the Window Manager. The Activity will be killed only when a low amount of the system memory is available.
- **Stopped:** when an Activity is in the background, not visible to the user, but still its state is preserved by the OS, the Activity is in the *Stopped* state; it still has a chance to return in the foreground (i.e. in the *Started* state), but owning a low priority, when the OS needs to satisfy resource requirements of higher priority activities it could be killed.
- **Terminated:** in this state, the resources retained by the Activity are completely released and the corresponding memory space is freed.

A different callback method is called at the occurrence of a transition between two of these states:

- **onCreate():** is called as soon as the Activity is created; it is typically used by the programmer to prepare the UI Components.
- **onStart():** is called when the Activity is becoming visible to the user.
- **onRestart():** is called if the Activity was in the *Stopped* state, before it is started again.

- **onResume():** is called just before the Activity can be used the user.
- **onPause():** is called when the Activity is going into the background, but has not been killed yet.
- **onStop():** is called when the Activity is no longer visible to the user; if the Activity is in the *Paused* state and it's going to be killed, this callback method will not be called.
- **onDestroy():** is called before the Activity is destroyed, either because the activity is finishing or is being destroyed by the system because more resources are needed by Activities whit an higher priority. The programmer can distinguish between this two scenarios calling the *isFinishing()* method.

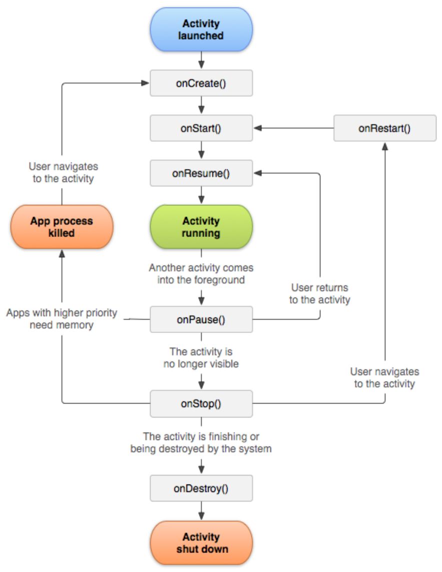

Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to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

Figure 2.3: Callback invocation sequences

Figure 2.3[5] shows the order of the callback invocations with respect to the Activity life cycle.

[5]http://developer.android.com/reference/android/app/Activity.html

2.1.2.3 Android User Interface Elements

User Interfaces in Android are built combining View and ViewGroup objects. A View is an object capable of drawing something on the screen and of capturing user interactions; common input controls such as text fields, buttons, labels and so on are subclasses of View. More View objects can be grouped together into a ViewGroup, that defines how the contained Views are arranged on the screen; the various layout models available in the Android Framework are subclasses of ViewGroup. A ViewGroup is also a subclass of View, so components can be easily nested using the *Composite pattern.* User Interfaces and UI Elements are therefore arranged into a hierarchy of View and ViewGroup objects.

The Android Framework provides some subclasses of View such as *Button*, that shows a button that can be clicked by the user to perform an action, *EditText*, that implements an editable text field, *Spinner:*, that shows a list from which the user can select an item; Figure 2.4[6] shows how they look like on the screen. Common used sublcasses of ViewGroup are *LinearLayout*, *RelativeLayout*, *TableLayout* and so on.

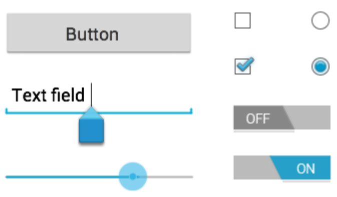


Figure 2.4: Android UI Input Controls

2.1.2.4 Android Fragments

Fragments are components of the Android UI that represent a behavior or a portion of UI in an Activity. They are like modules that can be reused on different

[6]http://developer.android.com/guide/topics/ui/controls.html

Activities and can be adapted accordingly to the screen size of the device. The *Adaptability* of a Fragment is better clarified by Figure 2.5[7] showing a typical example of use of Fragments. In this example, a Fragment contains a list of elements and the other the details about each element. On a larger screen both Fragments are shown together. On a smaller screen the Fragment containing the list occupies the whole screen; when an element is selected the other Fragment will be shown; pressing the BACK button will show the list Fragment again.

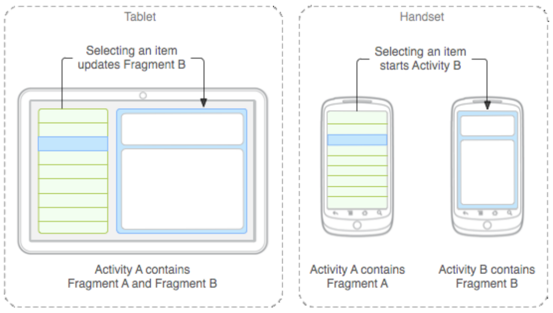

Tablet
Selecting an item updates Fragment B
Activity A contains Fragment A and Fragment B
Handset
Selecting an item starts Activity B
Activity A contains Fragment A
Activity B contains Fragment B

Figure 2.5: Fragment Adaptability

A Fragment has its own life cycle, directly affected by the one of the containing Activity and can be added, removed, replaced, hided or shown while the Activity is running.

The life cycle of a Fragment is shown in Figure 2.6[8]; as we can easily notice from the figure, the callback methods are in part similar to the ones of an Activity and are called in conjunction with those of the containing Activity. Differences can be found when a Fragment is initialized or destroyed. In particular, when a Fragment is added to an Activity the following callback are invoked:

- **onAttach():** the Fragment obtains a reference to the containing Activity, but neither the Fragment neither the Activity are fully initialized.

[7]http://developer.android.com/guide/components/fragments.html
[8]http://developer.android.com/guide/components/fragments.html

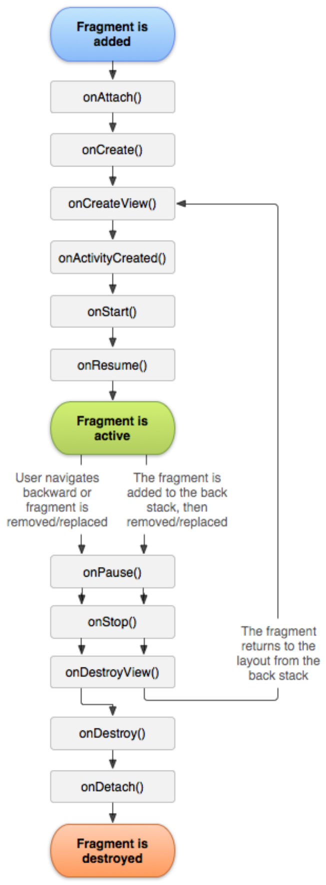


Figure 2.6: Fragment Life cycle

- **onCreate():** the Fragment is created by the OS.
- **onCreateView():** the UI of the Fragment can be initialized; a programmer should use this method to instantiate the components of the UI.
- **onActivityCreated():** the creation of the containing Activity has been completed; at this point the Fragment can interact with the Activity.

2.1.2.5 Android Services

Performing long-running operations in the thread of the User Interface is discouraged by the the official documentation of the Android SDK because it can reduce the responsiveness of the Application. To perform long-running tasks, the Android Framework provides the Service component. A Service runs in the background and does not provide direct user interactions; the user can interact with it through an Activity or another Service. Android Services can be **Started** or **Bound**. A *Started Service* continues running in the background also if the component that has initialized it has been terminated; this kind of Service can only be interrupted by a direct call. When a Service is **Bound** it acts like a server and lives only if there are connected clients. The Android Framework provides different System Services such as the **Location Service** that can be used to obtain information about the user location, the **SMS Service** that can be used to manage and send messages, the **Telephony Service** that can be used to manage phone calls, the **Sensor Service** that handles the communication of the application with the hardware sensors of the device.

2.1.2.6 Android Platform Security

The Android Operating System handles the problem of security exploiting the features offered by the Linux Kernel, using an approach similar to *sandboxing*. An Android application runs in an instance of a Virtual Machine (Dalvik or ART[9]), in a separate process of the Linux Kernel; so, the instance of an application and the memory it uses are completely isolated. In addition, both system and user applications run under a distinct system identity (User ID and Group ID), so applications are separated from each other and from the system ones.

An additional level of security is guaranteed through the *permission mechanism* that defines the access policies that grant certain privileges to the application that requests to execute a defined function of the Android API. Permissions requested by an Android application in the past versions of the Framework were defined

[9]https://source.android.com/devices/tech/dalvik/

by declaring them in the Manifest of the Application; as an example, Listing 2.1 shows the permission needed by an application to create a Socket. While the user is installing the application on his device, he is asked to grant all the required permissions; when the application is executed, it assumes that all the requested permission are granted.

Listing 2.1: Mainfest Permission Example

```
<uses-permission android:name="android.permission.INTERNET" />
```

Android M (Marshmallow, 6.0), the newest version at the time of writing, enforces a fine-grained access permission mechanism. Applications targeting Android M and above need to request their permissions at runtime. Permissions should still be declared in the Manifest, but the caller object, before accessing functions of the API that require a permission, should verify through the *ActivityCompat* object that the permission has been granted or request it to the user.

2.1.3 Android Testing Fundamentals

Android applications are tested using an extension of the jUnit Framework that provides the methods to interact with the Activities and GUI Objects of the AUT. An Android Application is typically composed by a set of Java classes that can be tested individually in the local JVM if they don't depend on the Android Framework Library. On the contrary, if the class under test exploits functions from the Android Framework Library, test cases should be installed together with the app on a virtual or a real device.

The *Instrumentation Framework* is the fundamental component of the Android Testing Framework and allows to control the life cycle, the state, the GUI and the running environment of the AUT. Figure 2.7 shows a subset of the classes composing the Android Testing Framework. **AndroidTestCase** extends directly the

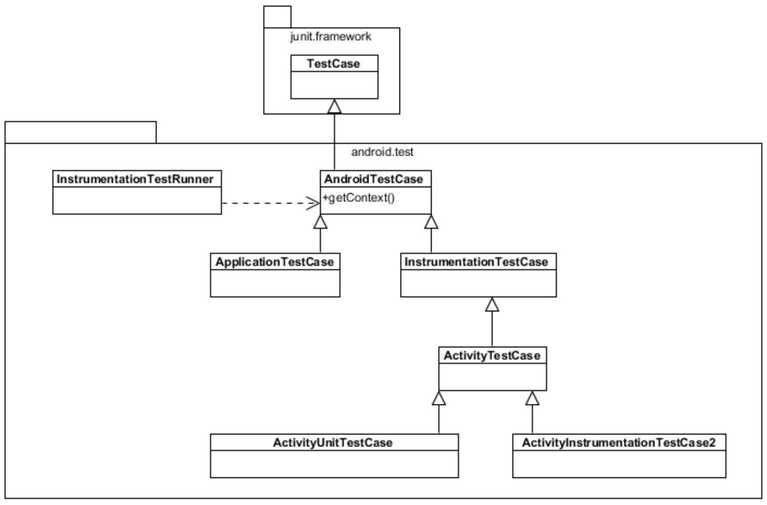


Figure 2.7: Android Testing Framework Overview

TestCase class of the jUnit Framework and to be run does not require the application to be executed; test cases extending the *AndroidTestCase* class can test components that do not need a GUI, like Services and Content Providers. **ActivityInstrumentationTestCase2** allows to perform functional tests of a single Activity, which can be accessed through the *getActivity()* method. The Activity under test is started and finished before and after each test, allowing to observe the evolution of the state of the GUI during the test execution. **ActivityUnitTestCase** provides an isolated environment for the execution of the Activity, which will be not connected to the system, that allows to have more control over the test environment of the Activity. **ApplicationTestCase** allows the test of the entire life cycle of the AUT. **InstrumentationTestRunner** is the class responsible of running the test cases.

2.1.4 Android Testing Tools

In the following some tools related to the functional testing of Android applications are briefly described.

Monkey[10] is a tool that runs on an emulator or a device able to generates pseudo-random sequences of user events such as clicks, touches, or gestures, as well as a subset of system-level events. **Monkey Testing Tool Library**[11] is a copy of the original Android Monkey tool, implemented on top of the Android Instrumentation for exploiting the random event generator in a test case.

As regards script-based functional testing there are two main frameworks available in the context of Android Applications. **Robotium**[12] is an extension of the Android Testing Framework created to ease the writing and understanding of script-based test cases. **Espresso**[13] is a test framework that allows the creation of automated script-based functional tests. Espresso features a simple and extensible API for the automation of the interaction with the GUI of the AUT; a test case using the Espresso framework behaves as if an actual user is using the app.

Robotium Recorder[14] is a Capture/Replay tool for Android Applications that support script-based functional testing; it generates test cases exploiting the Robotium Framework.

To support the execution of Android test cases the Android SDK provides some tools. As an example, **monkeyrunner**[15] that exposes an API for writing programs that control an Android device or emulator from outside of Android code. **Emma**[16] is a tool to measure code coverage in the context of Java applications. It allows to know how much and which parts of the source code have been actually exercised by the test case.

Finally, **Robolectric**[17] is a unit test framework that rewrites the Android SDK

[10]http://developer.android.com/tools/help/monkey.html
[11]https://code.google.com/archive/p/androidmonkey/
[12]https://github.com/robotiumtech/robotium
[13]http://developer.android.com/tools/testing-support-library/index.html#Espresso
[14]http://robotium.com/products/robotium-recorder
[15]http://developer.android.com/tools/help/monkeyrunner_concepts.html
[16]http://emma.sourceforge.net/
[17]http://robolectric.org/

classes as they are being loaded and making it possible for them to run on a regular JVM.

2.2 GUI Testing

Mobile applications are GUI-based applications, i.e., apps that have a GUI front-end. A GUI responds to user events, such as mouse movements or menu selections, providing a front end to the underlying application code. The GUI interacts with the underlying code through messages or method calls [75]. A well-known approach for testing GUI-based applications is through its GUI, by performing sequences of user input events on GUI widgets. This activity is known in the literature as GUI testing [76].

A key factor of any GUI testing technique is the test data generation approach. According to the systematic mapping presented by Banerjee et al. [21], there are several test data generation techniques usable in GUI testing, such as Capture/Replay, Model-based and Random testing. Other less popular methods are symbolic execution, formal methods, or statistical analysis.

Capture/Replay techniques record user interactions and convert them into test scripts that are able to automatically replay the interactions between user and application [49]. These techniques are usually exploited to perform regression testing and are very popular in several fields, such as Web application and desktop application testing [35].

Model Based techniques propose a different approach for test data generation, since they rely on a model of the GUI to generate test cases. According to Banerjee et al. [21], most common GUI models exploited for GUI testing are event flow graphs (EFG) and finite state machines (FSM). Usually these models have to be abstracted from the subject application, by executing resource-intensive activities.

In order to support the execution of this activity, several techniques and tools have been proposed in the literature. Memon et al. [73] present GUI Ripping, a dynamic process in which the GUI of the software is automatically *traversed* by opening all its windows and extracting all their widgets (GUI objects), properties, and values. The information gathered through this process can be exploited for generating test cases. In a later work, Memon et al. [74] redefined GUI Ripping as a technology that takes as input an executing GUI-based application and produces, as output, its workflow model(s). The technique has been implemented by Memon et al. in a tool within the *GUI Testing frAmewoRk* (GUITAR) [43], [74], [90]; the tool is not only able to reverse engineer the GUI of the application, but is also capable of automatically generating and executing test cases. Ana Paiva et al. [94] proposed a process that mixes manual with automatic exploration to reverse engineer structural and behavioral formal models of a GUI application. The goal is to diminish the effort required to construct the model and mapping information needed in a model-based GUI testing process. Analogously, Mesbah et al. [77] present a Web crawling technique that automatically detects and exercises all the elements of the Web application front-end that are capable of changing the state of the UI; the product of this crawl-based technique is a state-flow graph model that is composed by the states of the UI encountered during the crawling and the possible transitions between them. This model can be exploited for comprehending the AUT or for automatic generation of test cases of the AUT. Griebe et al. [42] present a model-based approach for testing context-aware mobile applications, based on a context-enriched UML Activity Diagram system model defined at design-time.

Another well-known approach of test data generation is Random testing. According to Hamlet [45] it is the simplest technique to select test cases. They are chosen at random from the input domain based on some distributions. Random testing is very popular in several fields, such as hardware testing, protocol testing, etc. In the field of GUI based applications, there has also been a growing interest in Random testing tools, also known as Monkey tools, that are used to perform crash testing of software applications. A monkey tool is able to test a program by sending it unstructured random input [78], implementing a kind of *fuzz testing*. Fuzz testing has shown its potential in assessing the robustness of software, while at the same time not requiring much effort for implementation.

2.2.1 GUI Testing Android Applications

As regards GUI Testing in the context of Android Applications, an extensive Systematic-Mapping-like search [101] [95] has been performed to find previous work on GUI Testing of Android Applications. In the first place, the ACM[18], IEEE[19] and Scopus[20] digital libraries have been considered and searched using the generic query *android testing* that produced 4192 entries. The result has been filtered by defining simple exclusion and inclusion criteria; in detail have been considered, articles, without duplicates, written using English language, related to *Engineering* and *Computer Science*, classified as *Paper* or *Journal Article* or *Book Chapter*, that in the abstract contained references to the presentation of one or more testing techniques for Android applications. Finally, to not compromise the quality of the search, each paper has been manually examined and filtered the ones related to *Automated GUI Testing Techniques in the context of Android Applications.* At the end of the whole process a list of 39 papers has been obtained. Figure 2.8 shows the distribution among the years of the selected papers, underlining how the interest in the Android GUI Testing techniques has grown over the last years (until November 2015).

In the following the contributions described in the selected papers are briefly

[18]http://dl.acm.org/

[19]http://ieeexplore.ieee.org/

[20]http://www.scopus.com/

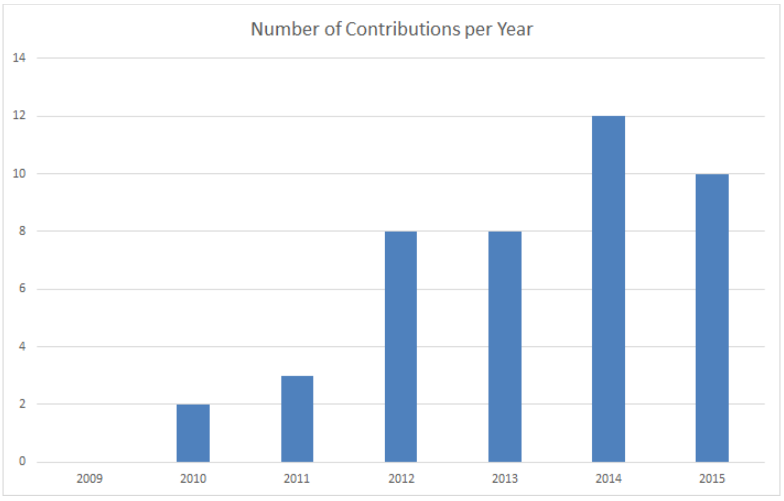


Figure 2.8: Contributions about Android GUI Testing per Year

summarized, excluding the contributions discussed in this thesis; they are ordered by the year of publication.

Liu et al. [66] propose *Adaptive Random Testing (ART)* a process based on an adaption of the FSCS-ART technique proposed by Chen et al. [26], [25] for testing of mobile applications. The sequences of events are randomly generated considering both GUI events and context events even if they are not actually handled by the application under test. The approach is implemented within the tool MobileTest.

Chang et al [24] present a platform independent approach to capture test visually and produce test script that uses images to specify which GUI components to interact with and what visual feedback to be observed.

Takala et al. [107] propose a set of tools for test modeling, design, generation and debugging, that use a Labeled State Transition Machine, belonging to the family of Finite State Machine models.

Hu et al. [52] present an approach based on the Monkey tool[21] that generates

[21]http://developer.android.com/tools/help/monkey.html

and sends a predefined number of random events to the application by means of the Monkey tool, in order to find different kinds of bugs in Android applications.

Amalfitano et al. [9] propose a technique based on a crawler that automatically builds a model of the GUI of the AUT; this model can be exploited to obtain test cases that can be automatically executed.

Nguyen et al. [91], describe a technique that combines model-based and combinatorial testing approaches that is based on a Finite State Machine model of the app; the technique is implemented in a tool called M[agi]C.

Zheng et al. [124] describe a method that combines static analysis and dynamic analysis to reveal UI-based trigger conditions. The dynamic analysis is used to enforce the execution along the suspicious path obtained from static analysis of the source code of the application.

Mirzaei et al. [82] propose a technique that exploits a symbolic execution technique implemented in the context of Java applications to generate test inputs for Android applications.

Anand et al. [14] present a fully automatic and general approach based on concolic testing, implemented in a tool called ACTEve.

Amalfitano et al. [12] [10] propose an automated technique that test an android application through its GUI: the GUI of an AUT is explored automatically with the aim of exercising it in a systematic manner.

Zhang et al [122] present a technique with the aim of finding invalid thread access errors in multithread Android applications.

Kaasila et al. [57] describe an online platform for executing UI tests on different physical devices based on a Capture/Replay technique. Test cases are manually recorded by the tester, then the test is uploaded to the platform that runs the test cases on a set of physical Android devices and reports the results back to the developer.

Gomez et al. [41] present RERAN, a technique and a tool for recording and replaying user interactions on the basis of information captured by examining the low-level event stream of an Android application.

Jensen et al. [55] propose a testing process that combine concolic execution and model-based techniques, implemented in a tool called Collider.

Rastogi et al [97] present AppsPlayground an automated dynamic security analysis of Android applications; it integrates the implementation of different techniques to come up with an effective analysis environment able to evaluate Android applications.

Yan et al. [117] propose an approach for testing for resource leaks in Android applications. Test case generation is based on a GUI Model obtained by combining the one built by means of AndroidRipper [9] and the information extracted manually from the source code. The approach is implemented in a tool called LeakDroid.

MacHiry et al. [68] present a random testing approach based on a *observe-select-execute* cycle implemented in the Dynodroid toolset. The authors present three different random techniques: *Frequency*, *Uniform Random* and *Biased Random*. The Frequency technique selects an event that has been selected least frequently by it so far. The Uniform Random technique selects an event uniformly at random. The Biased Random technique randomly selects an event also by taking into account the contexts the events belong to.

Azim et al. [17] propose two testing techniques implemented in the A^3E tool. One of the techniques, exploits a static analysis technique that analyze the byte-code of the AUT in order to infer a model called *Static Activity Transition Graph (SATG)*; this model is then used to generate test cases. The other technique, instead, is based on a dynamic analysis technique that automatically explores the GUI of the AUT by firing event in a depth-first manner and is able to infer a

Dynamic Activity Transition Graph, (DATG) model of the AUT.

Choi et al. [27] present two testing techniques that use machine learning to to generate user events sequences, based on the *Extended Deterministic Labeled Transition System (ELTS)* model of the GUI. Both techniques are based on the L* algorithm [15]: one is an implementation of this technique, the other is an improvement of the algorithm aimed at minimizing the number of the restarts of the application. These techniques are implemented in a tool called *SwiftHand*.

Yang et al. [118] propose a grey-box testing technique implemented in a tool called *ORBIT*. In a first phase, a model of the UI of the AUT is obtained via static analysis of its source code. Then, based on the inferred model, an automated exploration is performed by a crawler implementing two distinct strategies, respectively called Depth First Standard (DFS) and Crawling With Backtrack.

Liu et al. [65] describe an approach to capture user interactions and to generate test scripts for the replay phase; their approach supports assertions that can be used to catch errors in the execution of the generated test cases. They have implemented their approach in a tool called Android Capture and Replay testing Tool (ACRT).

Ying-Dar Lin et al. [63] present Smart Phone Automated GUI (SPAG) a Capture/Replay tool that runs on device, is based on an image processing techniques and dynamically changes the timing between events to adapt to the workload of the device; they present also an improved version called SPAG-C [64] that reduces the time of the testing process and increases the usability compared to the previous version.

Li et al. [62] propose a technique that exploits app executions recorded by the users (i.e testers) to spot during the replay phase certain *stop points* (i.e. points where the user stops for choosing next actions) and use them to guide a systematic exploration of the UI of the AUT. The systematic exploration is carried out by using techniques presented in literature both Random [52] and DFS [17] [118].

Zaeem et al. [121] propose a framework able to generate both test sequences and the corresponding assertions using an extensible library of oracles; the framework exploits a Finite State Machine model of the AUT and is implemented in a tool called QUANTUM.

Mahmood et al. [69] present a technique that automatically extracts from the code of the application two models, e.g., the Interface Model and the call Graph Model; these models are exploited for generating test cases by means of evolutionary techniques. The results of test case executions are then evaluated using a fitness function that rewards code coverage and uniqueness of the covered paths. The technique is implemented by the EvoDroid tool.

Van der Merwe et al. [110] present JPF-Android, a model checking tool for Android applications that allows them to be verified on Java PathFinder (JPF); the application is executed on a model of the Android software stack; the user and system input events are simulated for driving the application execution.

Wang et al. [112] propose a technique implemented in a tool called Droid-Crawler, that is that is very similar to the one presented by Amalfitano et al. in [10]. It performs an automatic exploration of the AUT using a depth-first strategy and infers a GUI Tree model of the GUI.

Hu et al. [53] describe an uniform random technique and a model-based technique that are able to automatically generate sequences of relevant events for the AUT. These techniques are implemented in a tool called *AppDoctor*. This tool also features a Capture/Replay technique.

Maya et al. [70] propose a technique for identifying data races in Android Application that is based on the systematic exploration of the UI of the tested application. The technique is implemented in the *DROIDRACER* tool.

Amalfitano et al. [6] propose a process to improve the effectiveness of test cases generated by the GUI Ripping technique presented in [10] using the model-based technique described by Marrone et al. in [72] [40].

Hao et al. [47] present a framework, called PUMA, that can be exploited for implementing several testing techniques by using a scripting language. In the paper the authors implement a pseudo-random testing technique for testing Android applications.

Mirzaei et al. [81] present a framework for automated testing of Android applications that automatically extracts models of the behavior and interface of an application and combine model-based testing with symbolic execution to systematically generate test cases.

Amalfitano et al. [11] propose a testing approach based on the GUI Ripping process [74]; the approach is based on three separate steps of Ripping, Generation and Execution. In the Ripping step, the GUI of the application is traversed and a FSM model of the GUI is constructed. The Generation step uses the generated model and a test adequacy criteria to obtain test cases, each modeled as a sequence of GUI events. In the Execution step test cases are replayed.

Espada et al. [99] propose a formal definition of a state machine that models the expected user interaction with the mobile application and a method to employ the model checker SPIN [50] to produce a set of test cases that generate traces for runtime verification tools. They implemented modeling and test generation phases in a tool chain called DRAGONFLY.

Morgado et al. [84] present an approach implemented in the *iMPAcT* tool that reverse engineers a mobile application in order to identify the UI Patterns of the AUT and test if they are correctly implemented. The process is based on an uniform random event generation technique.

Zhauniarovich et al. [123] present a random testing technique relying on the Monkey tool. The technique is implemented in the *BBoxTester* framework.

Wen et al. [114] propose a process based on the parallel execution of distributed testing nodes, coordinated by a centralized controller. The AUT is analyzed dynamically and test cases are generated by the nodes. The process is implemented in a tool called *Parallel Android Testing System (PATS)*.

CHAPTER 3

Android Ripper

In this chapter the *AndroidRipper* tool is described; it is based on the GUI Ripping technique and is able to automatically explore Android applications by exercising their GUI, to generate executable test cases, to abstract models of the GUI and to report the amount of source code covered by the generated test cases.

3.1 Introduction

Mobile applications are Event-Driven systems that can respond both to user-generated events (e.g. touch) and events generated by the system (e.g. generated by the sensors). The automatic testing processes for Event-Driven Software can be basically divided into two distinct phases:

- **Test Case Generation:** in this phase test cases are produced as sequences of one or more events.
- **Test Case Execution:** in this phase test cases are executed automatically i.e. the sequence of events that compose each test case is fired on the application under test.

Rothermel et al. [18] claim that there are two possible ways to combine these steps, summarized in Figure 3.1:

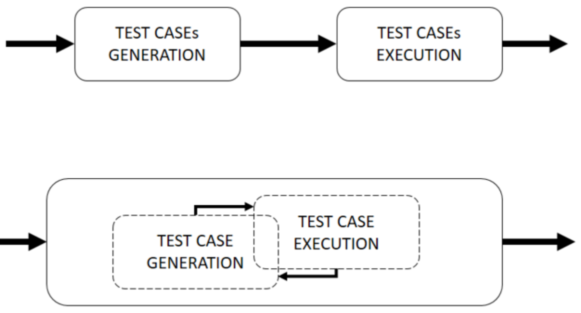


Figure 3.1: GUI Testing Automation Mechanisms

- ***Offline* test cases generation:** the two phases are separated and sequential; for example, in the model-based testing process, tests are generated from a model and subsequently executed; the same applies to techniques based on Capture/Replay, Symbolic Execution and so on.

- ***Online* test cases generation:** the two phases are not separated, but the sequences of events are extracted during the generation phase and then executed while the application is running, until a specific termination condition is verified.

In a study conducted by Amalfitano et al. [7] on automated testing techniques, the authors distinguish between Random testing techniques [88] and Active Learning testing techniques [96] [27]. Random testing techniques generate and execute pseudo-random sequences of events on the AUT [78]. Active Learning Testing Techniques combine Model Learning and GUI Testing Techniques [27]; these techniques are based on the learning of a model of the GUI of the AUT by which they generate sequences of events. The authors also show that there is a certain similarity in the way these techniques operate. In particular, they are based on the *exploration* of the application: iteratively they plan and trigger sequences of events on the GUI of the AUT until a termination criterion is not satisfied. These common characteristics were collected and summarized in Algorithm 1.

Algorithm 1 Unified Online Testing Algorithm

```
Require: TerminationCriterion, ExplorationStrategy, AbstractionStrategy, ExtractionCriterion, SchedulingStrategy
 1: if ($ExplorationStrategy == ActiveLearning$) then
 2:     $AppModel \leftarrow initializeAppModel(AbstractionStrategy)$;
 3: $stopCondition \leftarrow EvaluateStopCondition(TerminationCriterion)$;
 4: while ($!stopCondition$) do
 5:     $fireableEvents[] \leftarrow ExtractEvents(ExtractionCriterion)$
 6:     $eventsSequence \leftarrow ScheduleEvents(fireableEvents[], SchedulingStrategy)$;
 7:     $RunEvents(eventsSequence)$;
 8:     if ($ExplorationStrategy == ActiveLearning$) then
 9:         $AppModel \leftarrow RefineAppModel(AbstractionStrategy)$;
10:     $stopCondition \leftarrow EvaluateStopCondition(TerminationCriterion)$;
```

The algorithm requires as input five parameters that characterize a particular technique implementation:

- **TerminationCriterion:** defines the conditions for the termination of the process;
- **ExplorationStrategy:** specifies the type of the online technique, namely *ActiveLearning* or *Random*;
- **AbstractionStrategy:** defines the technique to analyze and describe the current status of the GUI of the AUT;
- **ExtractionCriterion** defines the criterion for the selection of the events that can be fired on the GUI of the AUT;
- **SchedulingStrategy:** specifies the strategy to define the next event to be triggered between the possible ones.

As for the functions called in the considered algorithm:

- **initializeAppModel:** if the algorithm belongs to the category of *Active Learning* testing techniques, this function initializes the model of the GUI of the AUT;
- **EvaluateStopCondition:** this function evaluates the *TerminationCriterion*
- **ExtractEvents:** using the *ExtractionCriterion* this function extracts the set of events that are fireable on the current GUI;
- **PlanEvents:** this function chooses the next sequence of events to be executed by using the *SchedulingStrategy*
- **RunEvents:** this function executes the chosen sequence of events;
- **RefineAppModel:** if the algorithm belongs to the category of *Active Learning* testing techniques, this function updates the model of the GUI of the AUT by abstracting its current status.

3.2 System Architecture

This section presents the system architecture of Android Ripper. Android Ripper implements Algorithm 1, presented in the previous section. It is developed in Java and it is based on a *Master-Slave* model. In the following, before diving into the details of the implementation of Android Ripper, an overview about its high-level components is given, describing what they do and how they work together.

3.2.1 Overview

Android Ripper is a tool for GUI testing of Android applications that is able to implement *Online GUI Testing techniques* (see Section 3.1). It is composed by three high-level components. Figure 3.2 shows the main components of Android Ripper and how they communicate.

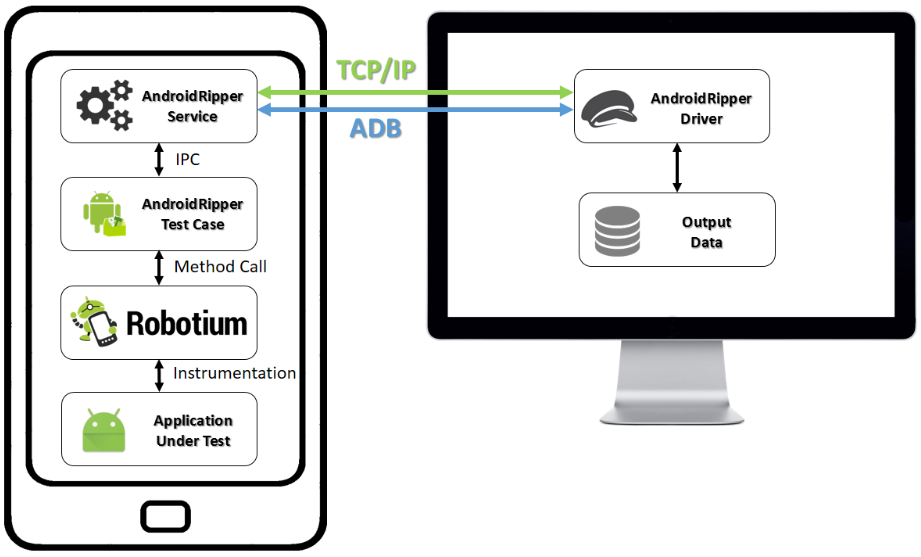


Figure 3.2: Overview of Android Ripper

Figure 3.2 shows that the *AndroidRipper Service* and *AndroidRipper Test Case* components are deployed on a virtual or real device running Android, while the *AndroidRipper Driver* runs in a Java Virtual Machine (JVM).

The **AndroidRipper Service** component is an *Android Service* (see Subsection 2.1.2.5) running in background on the target device that is responsible to mediate the communication among the *AndroidRipper Driver* and the *AndroidRipper Test Case* and to run operations that require specific permissions. The *raison d'etre* of this component resides in the *Android Permissions Mechanism* (see Subsection 2.1.2.6). It, in fact, solves the problem to create and use a TCP/IP Socket and to call other APIs of the Android Framework that need an explicit permission to be used without modifying the AUT. The TCP/IP connection is needed to communicate with the *AndroidRipper Driver* component; these messages are forwarded to the *AndroidRipper Test Case* via IPC.

The **AndroidRipper Test Case** component is an *ActivityInstrumentationTestCase2* jUnit test case (see Subsection 2.1.3) for the AUT that is responsible to execute events on the AUT and to abstract a description of its GUI. It exploits the *Robotium Library*[1] to interact with the AUT and the *AndroidRipper Service* to communicate with the *AndroidRipper Driver* by exploiting Android IPC Mechanism[2].

The **AndroidRipper Driver** component *drives* the execution of the testing process. It implements the main business logic of the tool and coordinates all the other components. The methods of *AndroidRipper Driver* implement the functions listed in Algorithm 1; they rely on the *AndroidRipper Test Case* for the execution of the scheduled events on the AUT and the retrieval of a description of its GUI.

Finally, the **AndroidRipper Installer** tool has been implemented to build and install *AndroidRipper Service*, *AndroidRipper Test Case* and the AUT on the target device; this tool exploits *Android SDK Tools*[3] and *Apache Ant*[4].

Figure 3.3 shows an overview of the packages of Android Ripper; these packages are further described in detail in the following section.

3.2.2 AndroidRipper Model

The GUIs of the application in Android Ripper are abstracted according to the conceptual model shown in Figure 5.1.

According to this model, the GUI is composed of instances called *GUI Interfaces*; a GUI Interface is composed of a set of visual items called *Widgets*; each Widget is defined by a *Type* and some *Properties* with their *Names* and *Values*. Examples of Widget properties are the position on the screen, the identifier and

[1]https://github.com/robotiumtech/robotium

[2]http://developer.android.com/guide/components/bound-services.html

[3]http://developer.android.com/tools/help/index.html

[4]http://ant.apache.org/

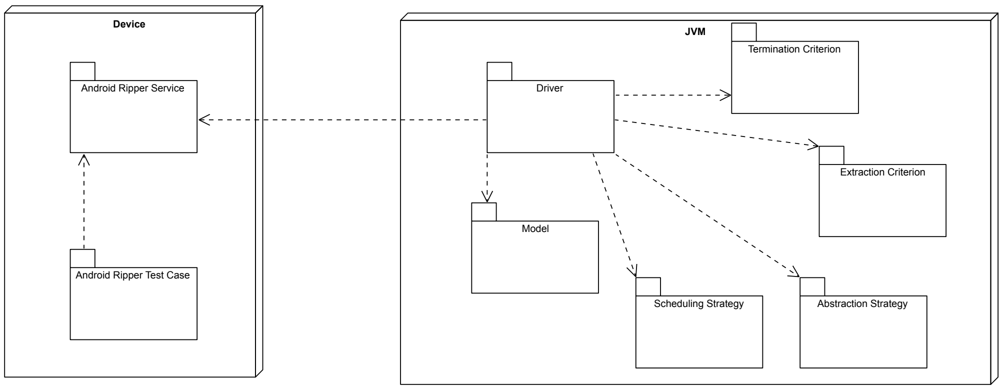


Figure 3.3: Package Diagram of Android Ripper

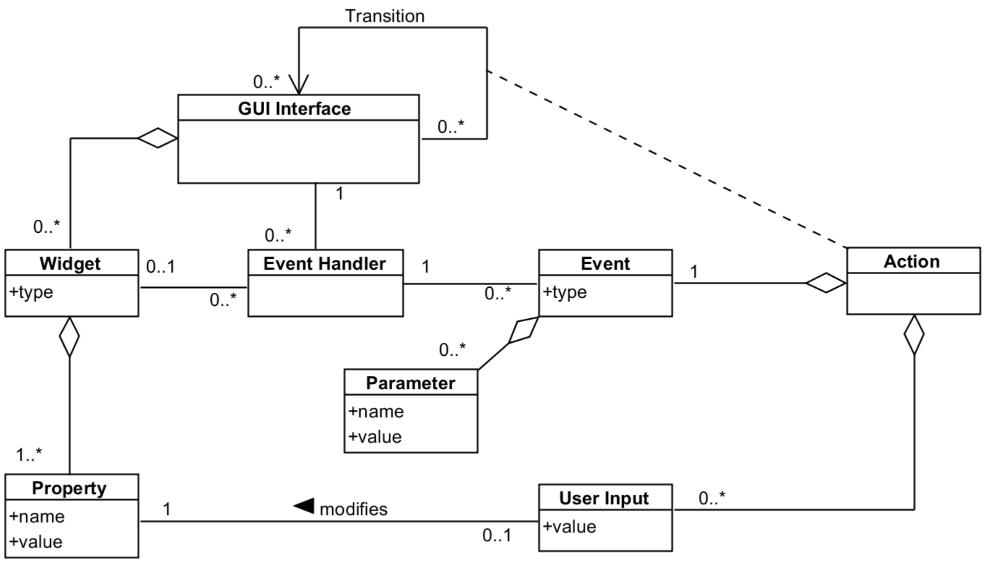


Figure 3.4: Conceptual Model of a GUI Interface

so on. *Event Handlers* are methods that can be defined in the context of a GUI Interface or directly in the context of a Widget and that are executed in response to the occurrence of an *Event.* Events may be *User Events* if they are triggered by a user interaction on the GUI (e.g. the tap on a button), or *System Events* if they are triggered by the execution environment (e.g. a pausing of the application). An Event may have zero or more *Parameters*; each Parameter is identified by a *Name* and a *Value.* As an example, parameters of a tap event are the coordinates of the point of the GUI Interface where the tap is performed by the user. An *Action* is composed by an Event and one or more *User Inputs* and is able to trigger a *Transition* between two GUI Interface instances (not necessarily different between them). An User Input consists of the modification of a Value of a property (e.g. the insertion of a text in a editable text field) that does not cause the execution of any event handler (elsewhere it is modeled as an Action).

On the basis of these definitions, a *Test Case* t is a sequence of pairs $\{G, A\}$, where A is an Action that can be performed on the GUI Interface G and that may generate the GUI Interface of the next pair of the sequence. The first pair starts from the Home interface of the app G_0. A Test Case can be also defined as $t = (G_0, A_0, ... G_m, A_m)$, where G_0 is the Home interface. A set of such sequences is a *Test-Suite.*

3.2.3 AndroidRipper Driver

Figure 3.5 shows an overview of the main components of the *Driver* package.

The classes in the package implement Algorithm 1. The **ActiveLearning-Driver** class implements an Active Learning version of the unified algorithm, while the **RandomDriver** class implements a Random version; both extend the **AbstractDriver** abstract class that contains the common methods.

The driver component depends on realizations of the *ExtractionCriterion*, *Schedul-*

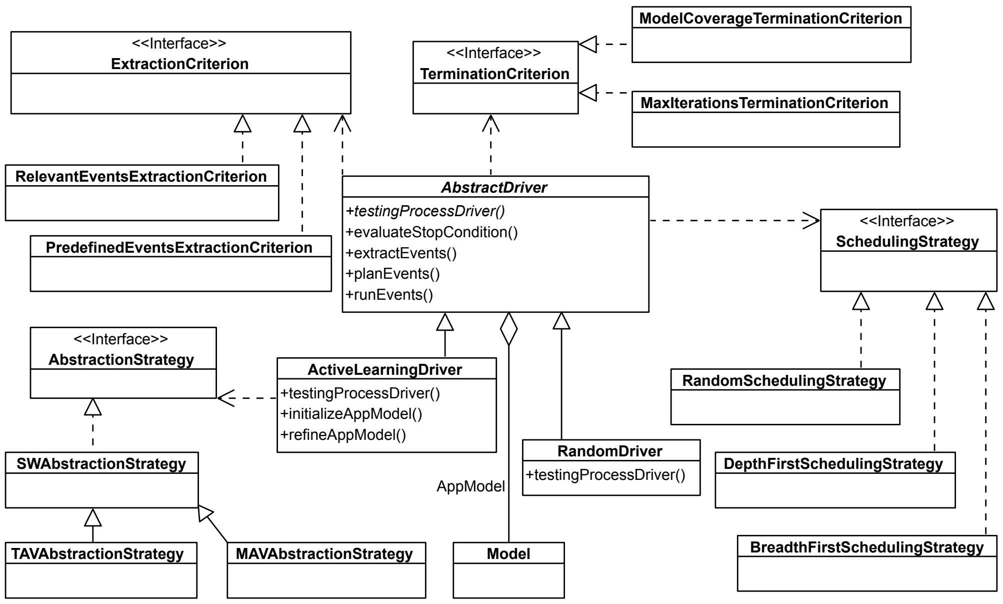


Figure 3.5: AndroidRipper Driver Package

ingStrategy, *TerminationCriterion* and *AbstractionStrategy* interfaces, that are related to the parameters required by Algorithm 1. The implementation features a library of classes implementing such interfaces.

A class implementing the **ExtractionCriterion** interface extracts the events that the technique can fire on the GUI of the AUT by exploiting the Model; in the library two implementations of this interface are proposed: the *PredefinedEventsExtractionCriterion* class that filters only a subset of predefined types of events and the *RelevantEventsExtractionCriterion* class that extracts relevant events only, e.g. events that can be actually handled by the app in its current state [68].

A realization of the **SchedulingStrategy** interface implements an algorithm for choosing the sequence of events that will be fired; three strategies are implemented in the library: the *RandomSchedulingStrategy*, that chooses the next event sequence in a pseudo-random manner and *BreadthSchedulingStrategy* and the *DepthSchedulingStrategy* that respectively schedule events using First-In-First-Out (FIFO) and Last-In-First-Out (LIFO) strategies.

A **TerminationCriterion** determines the approach used to stop the main loop of the algorithm; the library features two implementations: the *ModelCoverageTerminationCriterion* that terminates the process when no new events sequences can be generated on the learned model and the *MaxIterationsTerminationCriterion* that takes into account the number of iterations.

Finally, a class implementing the **AbstractionStrategy** interface determines the strategy implemented by a technique for abstracting the model of the GUI; the implementation features the *SWAbstractionStrategy* class that considers the set of widgets composing the GUI and the values assumed by its attributes. This class is further specialized in: *TAVAbstractionStrategy* that considers only the type attribute of the widgets and *MAVAbstractionStrategy* that considers also the values of others attributes of the widgets.

3.2.4 AndroidRipper Test Case

The main classes contained in the *AndroidRipper Test Case* package are shown in Figure 3.6.

The **RipperTestCase** class extends *ActivityInstrumentationTestCase2* (see Subsection 2.1.3); it is connected via IPC to the *AndroidRipper Service* and through its interfaces can send and receive messages to the *AndroidRipper Driver*. The (unique) test method *testApplication()* does nothing else than initialize the Robot, the Extraction and the Automation components and wait to be terminated by a command of the *AndroidRipper Driver*. Each time a command from the *AndroidRipper Driver* is received, the *RipperTestCase* handles it by calling the related component.

The **Automation** class is used when the *RipperTestCase* is requested to perform an event or fill an input field on the GUI of the AUT; the **Extractor** class, instead, is exploited to obtain the description of the current status of the GUI

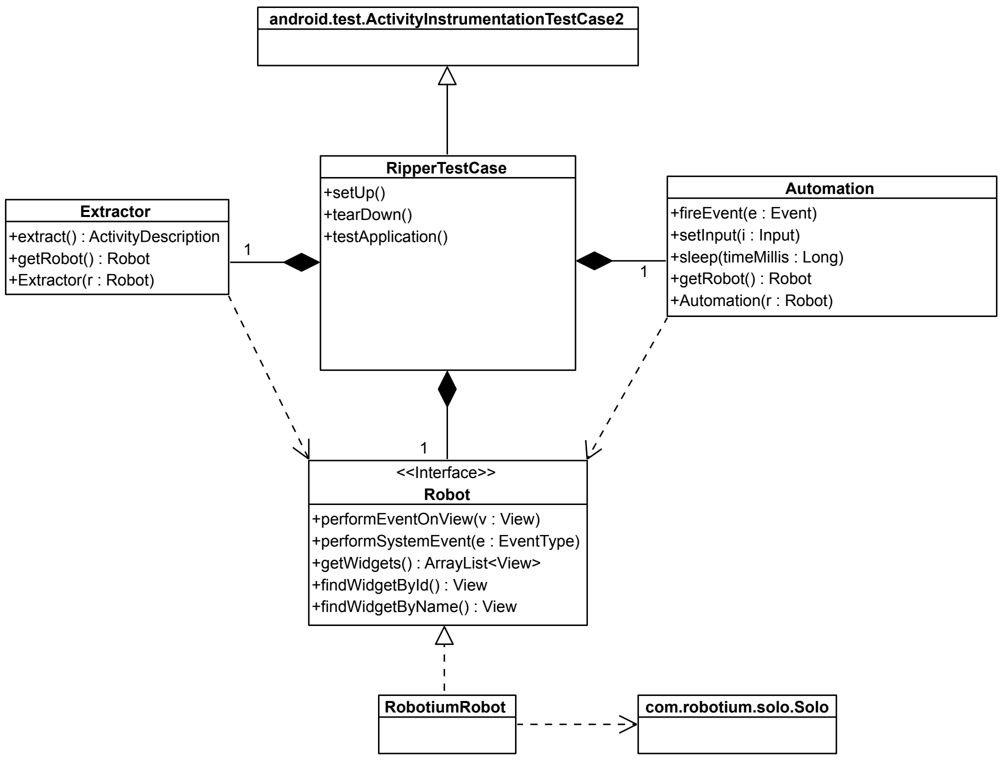


Figure 3.6: AndroidRipper Test Case Package

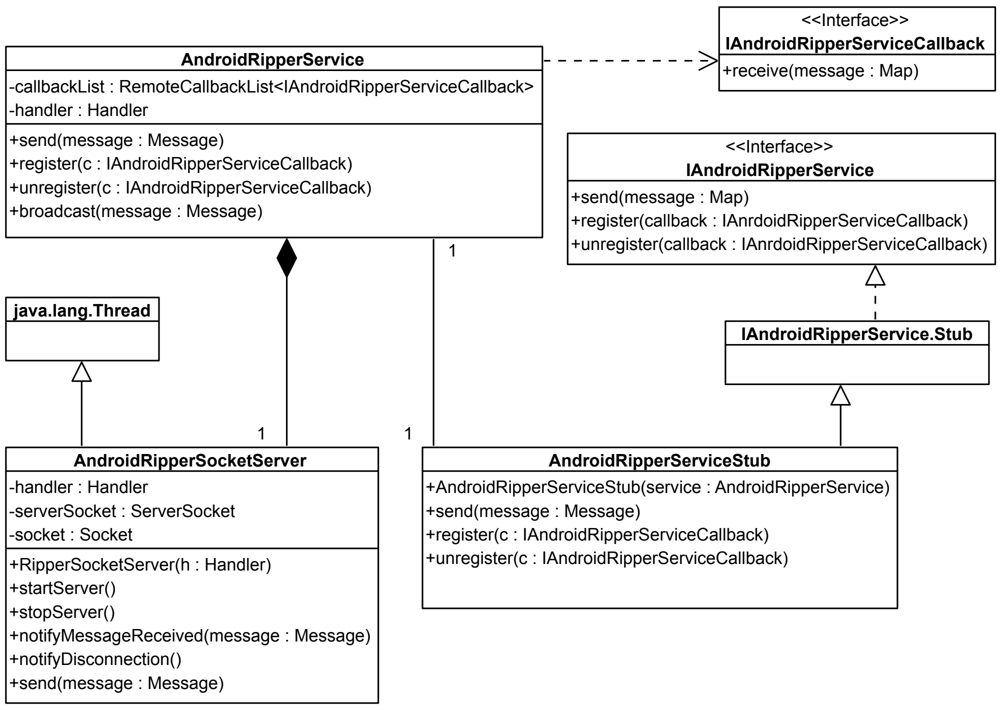
of the AUT. Both classes make use of the methods of an implementation of the **Robot** interface that is responsible of interacting with the GUI of the AUT and the running environment; our implementation exploits the functions offered by the *Robotium Library*[5].

3.2.5 AndroidRipper Service

Figure 3.7 shows the classes belonging to the *AndroidRipper Service* package.

This Android Service acts as a proxy between *Driver* and *AndroidRipper Test Case*. The **AndroidRipperSocketServer** class implements a TCP/IP server that listens for remote requests from the *Driver*. On the other side, the *Service* exposes the **IAndroidRipperServiceCallback** and **IAndroidRipperService**

[5]https://github.com/robotiumtech/robotium



AndroidRipperService
-callbackList : RemoteCallbackList<IAndroidRipperServiceCallback>
-handler : Handler
+send(message : Message)
+register(c : IAndroidRipperServiceCallback)
+unregister(c : IAndroidRipperServiceCallback)
+broadcast(message : Message)
<<Interface>>
IAndroidRipperServiceCallback
+receive(message : Map)
<<Interface>>
IAndroidRipperService
+send(message : Map)
+register(callback : IAnrdoidRipperServiceCallback)
+unregister(callback : IAnrdoidRipperServiceCallback)
IAndroidRipperService.Stub
java.lang.Thread
1
1
1
AndroidRipperSocketServer
-handler : Handler
-serverSocket : ServerSocket
-socket : Socket
+RipperSocketServer(h : Handler)
+startServer()
+stopServer()
+notifyMessageReceived(message : Message)
+notifyDisconnection()
+send(message : Message)
AndroidRipperServiceStub
+AndroidRipperServiceStub(service : AndroidRipperService)
+send(message : Message)
+register(c : IAndroidRipperServiceCallback)
+unregister(c : IAndroidRipperServiceCallback)

Figure 3.7: AndroidRipper Service Package

remote interfaces that allow the *AndroidRipper Test Case* to be bound to the service for receiving requests and sending responses; these interfaces are defined using AIDL[6].

3.2.6 AndroidRipper Master-Slave Interaction

Figure 3.8 shows the interaction between *AndroidRipper Driver* (Master) and *AndroidRipper Test Case* (Slave).

AndroidRipper Driver implements the functions of Algorithm 1. To interact with the AUT this component exploits the *AndroidRipper Test Case*. When the *Driver* wants to update the *AppModel*, it asks for a complete description of the current GUI status of the AUT to *AndroidRipper Test Case* that calls the function

[6]http://developer.android.com/guide/components/aidl.html

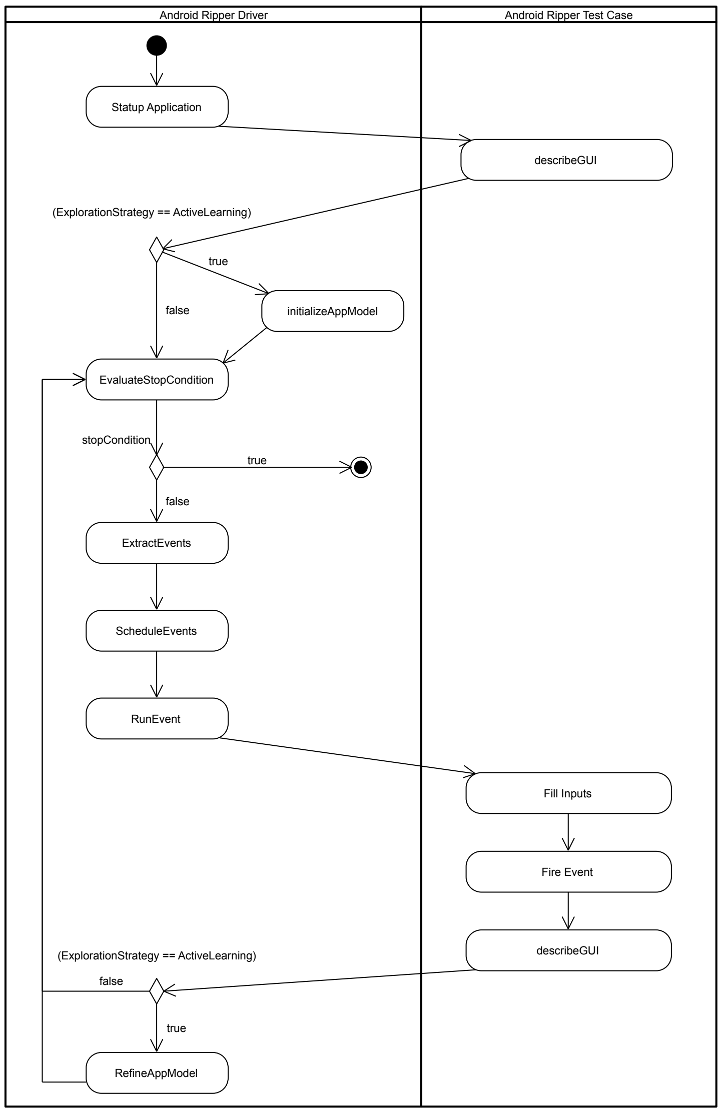


Figure 3.8: AndroidRipper Master-Slave Interaction

describeGUI() that returns the requested description. To execute an Action (see Subsection 3.2.2) on the AUT the *Driver* invokes the *AndroidRipper Test Case* that simulates *User Inputs* composing the Action by calling the method *fillInputs()* and fires the related *User Event* by executing the method *fireEvent()*; after the execution a description of the status of the GUI is returned to the *Driver*.

3.3 Case Study

3.3.1 Subject Application

The application selected as subject of the case study is Trolly[7]. Trolly is a simple and intuitive open source shopping manager application published on Google Play having more than 5 thousand of downloads. Trolly provides features for handling a list of shopping items. Each item can be created, added into the list, edited, deleted and changed in state. A shopping item can assume one of the following three states:

- *IN-LIST*: when the user has to buy it;
- *IN-TROLLEY*: when it has been already bought by the user;
- *OFF-LIST*: when it has been deleted from the list, but is still stored into the local database.

The Trolly source code is made of 19 classes, 3 packages, 64 methods and a total of 364 executable Java LOCs. The data persistence is guaranteed by a local SQLite database. Moreover, the application has a single activity class implementing the GUI. As shown in Figure 3.9 the GUI offers three widgets the user can interact with: an *Add an item* TextEdit, an *Add* Button and a List of items.

[7]https://play.google.com/store/apps/details?id=caldwell.ben.trolly

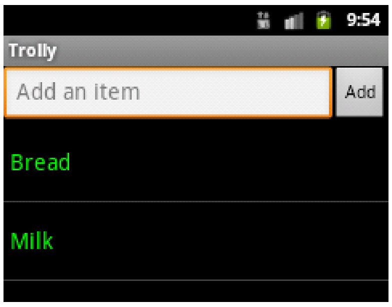


Figure 3.9: Excerpt of the initialized user interface

The user can fill the TextEdit with the name of the item he would like to buy, then he taps on the Button to add this item into the List. The new added item will be in the *IN-LIST* state. The remaining features provided by Trolly can be reached through the device menu or by means of the context menu that appears when a long tap event is fired on one of the items. Trolly provides two kinds of views. In the default mode view the *OFF-LIST* items are hidden, whereas in the adding mode view they are shown in the items list and have the dark grey color. The *IN-LIST* items are always rendered as green in the list. Moreover, Trolly offers two further functionality. The first one, called reset list, deletes all the items from the local database. The second one is an *Autofill* features that helps the user when he adds a new item by listing the names of the items stored into the database.

3.3.2 Metrics

The performance of a testing technique is evaluated in this experiment by measuring both the reached test adequacy and the cost for its execution.

The test adequacy is measured in terms of *LOCs coverage* that is the number of lines of the source code of the subject application that are executed by a testing technique. The *LOCs Coverage %* is the ratio between the *LOCs coverage* and the number of statements of the subject application, measured in percentage. The cost for the execution of each technique is evaluated in terms of the number of fired events (*# of Fired Events*). Moreover, to evaluate how much the choice of the parameters may influence the GUI model produced by Active Learning testing techniques, the complexity of the inferred GUI tree models has been measured. The considered Active Learning techniques reconstruct a model called *GUI Tree* where the nodes represent instances of the user interfaces in the Android application, while edges describe event-based transitions between interfaces [12]. Specific metrics are evaluated on these models, such as the number of nodes (*# of Nodes*), the number of edges (*# of Edges*), the number of leaves (*# of Leaves*) and the maximum depth of the GUI tree (*Depth*).

3.3.3 Experiment Setup

3.3.3.1 Application Preconditions

The Trolly application is initialized with two items in the List every time it is launched. Moreover, both the items are in the $IN - LIST$ state. Figure 3.9 shows an excerpt of the Trolly user interface that is initialized with the described precondition where the *Bread* and *Milk* items are in the shopping list and are in the $IN - LIST$ state.

3.3.3.2 Considered Testing Techniques

Both Active Learning and Random techniques are considered in the experiment. The Active Learning techniques feature a combination of the following values for the parameters of Algorithm 1:

- **ExtractionCriterion:** *RelevantEventsExtractionCriterion* (RE in the following)

Table 3.1: Considered Testing Techniques

	ExtractionCriterion	**TerminationCriterion**	**AbstractionStrategy**	**SchedulingStrategy**
T1	RE	MC	MAV	BF
T2	RE	MC	MAV	DF
T3	RE	MC	TAV	BF
T4	RE	MC	TAV	DF
T5	RE	MI	-	R

- **TerminationCriterion:** *ModelCoverageTerminationCriterion* (MC in the following)
- **AbstractionStrategy:** *TAVAbstractionStrategy* (TAV in the following), *MAVAbstractionStrategy* (MAV in the following)
- **SchedulingStrategy:** *DepthFirstSchedulingStrategy* (BF in the following), *BreadthFirstSchedulingStrategy* (DF in the following)

While the considered Random technique features:

- **ExtractionCriterion:** *RelevantEventsExtractionCriterion* (RE in the following)
- **TerminationCriterion:** *MaxIterationsTerminationCriterion* (MI in the following), limited to 1000 iterations
- **SchedulingStrategy:** *RandomSchedulingStrategy* (R in the following)

The *ExplorationStrategy* parameter is implemented respectively in the *ActiveLearningDriver* and *RandomDriver* classes.

In detail, the considered techniques are reported in Table 3.1, where each combination is labeled as *Tx*.

3.3.3.3 Testing Environment

AndroidRipper was executed on a set of PCs, each one equipped with a Windows 7 operating system, 64 bit Intel I5 processor at 3GHz, and 4 GBytes of RAM. Trolly was executed and tested on the Android Virtual Device (AVD) provided by the Android Developer Toolkit (ADT). The AVD was configured for emulating a device having 512 MByte of RAM, a 64 MByte SD Card, and an Android Gingerbread (2.3.3) installed on it. Moreover, Trolly was instrumented through the Emma[8] tool for measuring the code coverage.

3.3.4 Results

Table 3.2, Table 3.3 and Table 3.4 report the results obtained from the execution of each testing technique.

As for the performances, Table 3.2 shows the reached testing adequacy values (in terms of LOCs coverage and LOCs percentage) whereas Table 3.3 the costs in terms of number of fired events. Regarding the reconstructed models, Table 3.4 reports the complexity metrics values evaluated for the GUI Tree models that were inferred by the six active learning testing techniques. As data show, the choice of the parameters strongly influences the results of the testing techniques. As for the performances, all the testing techniques reached different values of *LOCs Coverage* and only two couples of techniques fired the same number of events.

None of the techniques covered all the source code and they did not reach the 100% of *LOCs Coverage %*. The highest coverage percentage (294.4/364) was obtained by the Random technique $T5$, while the lowest (220.6/364) was obtained by the technique $T2$.

By fixing both the *SchedulingStrategy*, the effect of the *AbstractionStrategy* on

[8]http://emma.sourceforge.net/

Table 3.2: Test Adequacy Values

Testing Technique	LOCs Coverage	LOCs Coverage %
T1	223.3	(223.3 / 364) = 61%
T2	220.6	(220.6 / 364) = 61%
T3	268.3	(268.3 / 364) = 74%
T4	270.3	(270.3 / 364) = 74%
T5	294.4	(294.4 / 364) = 81%

Table 3.3: Cost Values

Testing Technique	# of Fired Events
T1	58
T2	58
T3	289
T4	244
T5	1000

the performances of the Active Learning testing techniques can be observed. Active Learning techniques implementing a MAV strategy reached higher testing adequacy values than the ones based on TAV. The coverage of $T3$ (74%) overcame the one of $T1$, and the coverage of $T4$ (74%) was higher than $T2$. On the contrary, techniques that reached lowest levels of testing adequacy were cheaper, since they fired less events. In effect, the ones based on TAV needed to trigger 58 events, while more than 250 events were sent by the techniques based on the MAV strategy.

Conversely, the *SchedulingStrategy* does not have a fundamental influence on the performances of the Active Learning techniques. Indeed, techniques based on

Table 3.4: GUI Tree Complexity Metrics Values

Testing Technique	**# of Nodes**	**# of Edges**	**Depth**	**# of Leaves**
T1	59	58	5	51
T2	59	58	7	51
T3	290	289	11	255
T4	245	244	16	215

the same *AbstractionStrategy* obtain very similar results as the *SchedulingStrategy* varies. As an example, $T1$ covered only 3 lines of code more than $T2$, and they fired the same number of events. Analogously, $T2$ covered two LOCs less than $T4$. Regarding the $T5$ technique, it is the better technique in terms of test adequacy since it is able to cover more code than any of the other considered techniques, but it needs more than double than the number of events triggered by the two MAV based techniques and more than 10 times the number of events triggered by the two TAV based techniques.

As for the complexity sizes of the GUI Tree models inferred by the techniques, they are fundamentally affected only by the adopted *AbstractionStrategy*. Techniques $T1$ and $T2$, implementing the TAV strategy obtained GUI trees less complex than the ones inferred by $T3$ and $T4$ that are both based on the MAV. The *SchedulingStrategy* does not always influence the complexity of the models.

Techniques $T1$ and $T2$ learned GUI trees having the same number of nodes, edges and leaves, but $T2$ inferred a deeper GUI tree. Figure 3.10 and Figure 3.11 show the models that were learned by the $T1$ and $T2$ respectively. Eventually, technique $T4$ obtained a deeper GUI tree than the one inferred by $T3$, but with a lower number of nodes, edges and leaves.

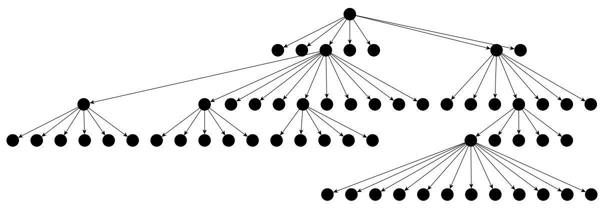


Figure 3.10: GUI Tree inferred by $T1$

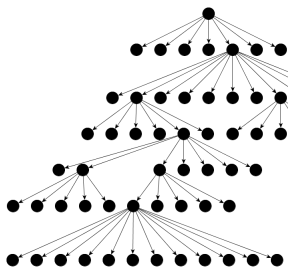


Figure 3.11: GUI Tree inferred by $T4$

3.3.5 Source Code Coverage Analysis

After this quantitative analysis, a detailed analysis was performed to examine the portions of code that were covered or not by the different techniques.

A static analysis of the code revealed that the application has no dead code. The code that is not covered by any of the techniques is due to the chosen preconditions. For example, the portion of code related to the creation of new database tables is never executed since the database already exists and the application does not provide features for database elimination. Moreover, there is a portion of code that is executable only if the application is executed starting from a specific entry point (via an Intent message with a specific parameter). Since in the experimentation any exploration is started by directly opening the GUI of the application, this portion of code cannot be executed. On the other hand, the limitations related to the selection of random input strings in the *SchedulingStrategy* do not cause loss of coverage in this application.

The Active Learning techniques based on the *AbstractionStrategy* TAV and MAV are not able to cover more than 20 LOCs that are covered by the Random technique $T5$. These lines are related to functions executed on items that are in the IN-TROLLEY or in the OFF-LIST states. They are not executed since TAV and MAV consider equivalent two GUI instances containing a list with the same number of items, without considering the colors of the items. For this reason, only the first GUI instance showing a list with items that are in these two states is deeply explored by firing events executable on it.

Moreover, the Active Learning techniques based on the MAV *AbstractionStrategy* (i.e. $T3$ and $T4$) covered more than 40 LOCs more than the corresponding techniques based on TAV. In particular, the code related to the *Add* and to the *Autofill* functions is not covered by techniques based on TAV. The reason is that the GUI instance obtained after the insertion of a value in the text field is considered equivalent to the previous one by the TAV strategy that does not take into account the value written in the text field. The techniques based on MAV, instead, continue the exploration of this GUI Instance by clicking the *Add* button (causing the execution of the Add function) and by inserting a value in the text field (causing the call of the *Autofill* function).

The coverage reached by the Active Learning techniques based on the BF scheduling strategy (e.g. $T1$ and $T3$) differs of few lines with respect to the coverage reached with the DF strategy (e.g. $T2$ and $T4$ respectively). The differences in coverage between these two strategies are only due to the different order in the triggering of the events and to the different depth of the generated GUI trees. In example, code related to the state change from $IN-TROLLEY$ to $IN-LIST$ is covered only by the $T4$ technique, while code related to the state change from $OFF-LIST$ to $IN-LIST$ is covered only by the $T3$ technique. Moreover, the *AbstractionStrategy* TAV is not able to distinguish between two different dialogs related to the *Clear List* and to the *Reset List* functions. For this reason, the $T1$ exploration technique covers only the code related to the first operation while the $T3$ technique covers only the code related to the second operation due to the different order of execution of the events.

3.4 Conclusions

In this chapter AndroidRipper is presented as a possible implementation of the generic algorithm Algorithm 1. Using AndroidRipper, an experiment on a real Android application was carried out obtaining some interesting aspects to improve both the effectiveness and the efficiency of the presented testing techniques.

In particular, the presented *RelevantEventsExtractionCriterion* implementation only extracts events related to user's interactions, while an Android application can be also sensitive to system and hardware-related events. In Chapter 4 a technique that generates automatically some of these events in Android Ripper is presented.

Moreover, Active Learning techniques try to limit the size of the generated *AppModel* and to avoid the repetition of already executed sequences of events. On the other side, Random techniques that can generate redundant and longer sequences of events may reach an higher effectiveness but are generally more costly than Active Learning techniques. In Chapter 5 a Genetic Algorithm implementation is presented; this technique aims to increase the effectiveness of a test suite with respect to the one obtained by Active Learning techniques by introducing randomness in the sequences of events composing the test cases, while it tries to reduce the effort needed with respect of completely random techniques, .

As regards the cost of Random techniques, in Figure 3.12 is reported the trend of the LOCs Coverage obtained by executing the Random technique $T5$ on the

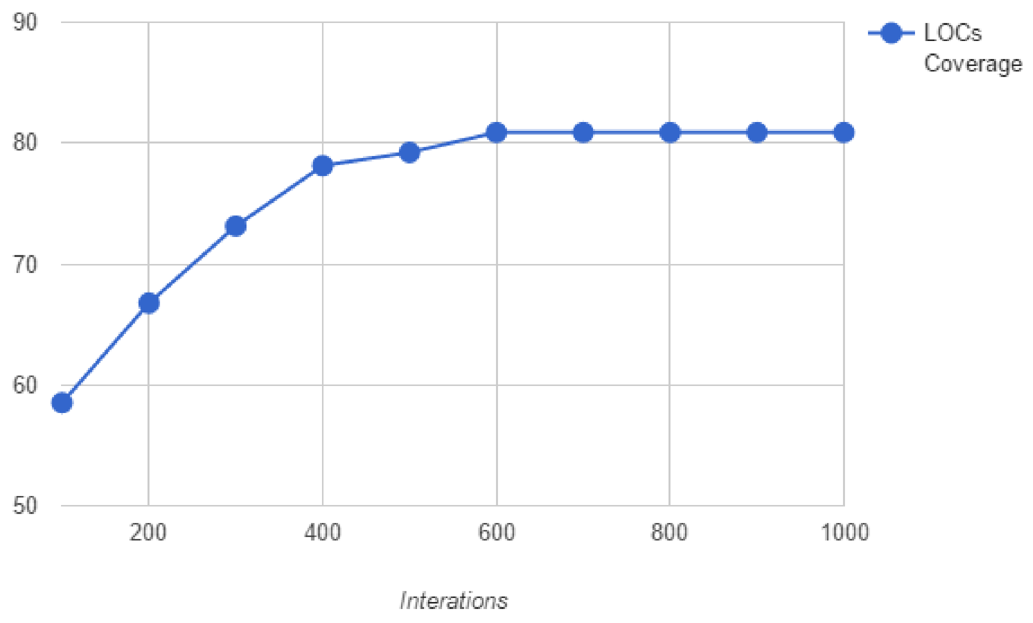


Figure 3.12: Trend of the LOCs Coverage for Trolly

Trolly application for increasing numbers of iterations of Algorithm 1.

In the case study, the Random technique was arbitrarily terminated after 1000 iterations, but the code coverage does not grow after the 600th iteration. So, 600 iterations could be a more efficient termination point. But, there is no guarantee that continuing the testing process none of the previously uncovered LOCs will be covered. In Chapter 6 is presented a criterion that addresses the problem of stopping a Random testing process at a cost-effective point, where test adequacy is maximized and no testing effort is wasted.

Finally, as regards the time needed for the execution of the testing techniques, for $T1$ and $T2$ that are based on the *TAVAbstractionStrategy* this time is about 1 hour, for $T3$ and $T4$ that are based on the *MAVAbstractionStrategy* is about 14 hours and for the Random technique $T5$ is about 30 hours. In Chapter 7 a parallel implementation of Algorithm 1 is presented with the aim of increasing the efficiency, reducing the execution time needed.

CHAPTER 4

Considering Context Events in Event-Based Testing of Mobile Applications

This chapter is focused on the problem of testing a mobile applications taking into account the context and context-related events. To this aim I describe:

- possible strategies of event-based testing that take into account contextual events;
- how event-patterns could be used in three scenario-based mobile testing approaches;
- some technological solutions for implementing the proposed scenario-based testing techniques in the Android platform;
- an example of using one of the proposed techniques for testing real Android apps.

The work described in this Chapter has been done in collaboration with the *REvERSE Research Group*[1].

[1]http://reverse.dieti.unina.it/

4.1 Introduction

Some specific characteristics of handled devices must be carefully considered when testing mobile applications. They include heterogeneity of hardware configurations of mobile devices, scarceness of resources of the hardware platform, and variability of their running conditions.

Heterogeneity of mobile device platforms (that come equipped with diverse hardware sensors, screen displays, processors and so on) implies the need for expensive cross-platform development and testing [61]. The scarceness of resources of the hardware platform requires specific testing activities designed to reveal failures in the application behavior due to resource availability (such as battery charge level, available RAM, wireless network bandwidth and so on). The variability of running conditions of a mobile app depends on the possibility of using it in variable contexts, where a context represents the overall environment that the app is able to perceive. More precisely, Abowd et al. [1] define a context as: "any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is including the user and applications themselves".

When a mobile application has been designed to be aware of the computing context in which it runs and to adapt and react according to it, it belongs to the category of context-aware applications [20]. These apps may be notified of any change to their context by means of events.

Context awareness of mobile apps yields several new challenges for mobile app testing too, since an app should be tested in any environment and under any contextual input [85]. However, a considerable part of mobile app testing literature omits to consider the context-awareness issue [9], rather focuses on specific mobile problems such as testing in variable network conditions [102], security testing [36], performance testing [59], or GUI testing [12][52]. Another part addresses context-aware testing issues [39][66][100][109][113] proposing solutions to the problem of context modeling and deriving test cases based on the proposed models.

4.2 Event-Based Testing Techniques for Mobile Apps

In event-based testing of event-driven systems (EDS), the behavior of the system is checked with input consisting of specific event sequences [23].

Mobile apps are event-driven systems, too, but, differently from other traditional event-driven software systems, like Desktop or Web applications, they are able to sense and react to a wide set of events besides user ones. Mobile devices are equipped indeed with a wide variety of hardware sensors that are able to sense the context in which the device stays and to notify context changes to the running app by means of events.

Therefore, since the user can be considered as a part of the context of an app [1], in event-based testing the application behavior should be checked in response to several types of context event, such as:

- user events produced through the GUI;
- events coming from the external environment and sensed by device sensors (such as temperature, pressure, GPS, geomagnetic field sensor and so on);
- events generated by the device hardware platform (such as battery and other external peripheral port, like USB, headphone, network receiver/sender and so on);
- events typical of mobile phones (such as the arrival of a phone call or a SMS message);

- events like the receiving of an e-mail or a social network notification, that are related to the fact that modern mobile phones are more and more "connected".

It is important to remark that not all the mobile apps are designed to react to GUI events. As an example, Muccini et al. [85] distinguish between MobileApps, that are mobile applications reacting to all contextual events (both GUI and non-GUI ones) to generate context-based outputs, and Apps4Mobile that react only to GUI events, like traditional applications that have been rewritten to run on mobile devices.

Mobile Applications belonging to the Apps4Mobile category may be effectively tested by using testing techniques designed for testing traditional applications, (like GUI based testing for desktop or Web applications [119]). MobileApps, instead, require event-based testing techniques that properly consider all types of context-related events. This testing activity may be very expensive due to the large number of possible contexts, event classes and combinations of events and contexts to be considered.

Effective strategies for test case generation should be used to define the sequences of events of mixed types. To this aim, both "simple" strategies not requiring any specific knowledge about the app under test, and more "systematic" approaches based on such knowledge are usable. As an example, simple approaches may define event sequences just trying to achieve the coverage of each class of contextual events with a fair policy. This technique may help to discover unacceptable behaviors of the app (like crashes or freezes) that are often reported in bug reports of mobile apps and appear when the app is impulsively solicited by contextual events like the ones notifying connection/disconnection of a plug (USB, headphone and so on), an incoming phone call, the GPS signal loss (for instance when the device enters a tunnel), and similar ones.

Other systematic test generation strategies may require the coverage of specific event sequences representing specific usage scenarios of the application.

Scenarios are software specifications defining relevant ways of exercising an application by sequences of events. They may be described by different formalisms, like MSC (Message Sequence Charts), State-Transitions systems, Event-Flow-Graphs (EFG), UML sequence diagrams and so on. There are several scenario-based testing approaches presented in the literature. As an example, in a work by Malik et al. [71] scenarios are derived from Event-B model [2] of the system under test (where an Event-B is a formalism for modeling the behavior of event-based systems as a state transition system) and transformed into executable JUnit test cases thanks to Java language implementation templates. Garzon et al. [39] presented an approach based on the DEVS formalism [111] (a discrete event system specification) to model scenario-specific event-patterns. Starting from the model, it is possible to produce automatically several event sequences that represent valid sensor-related traces of a given scenario by means of an event generator application. This model-based approach facilitates and speeds up the generation of sensor-based data for context-aware applications testing. Wang et al. [113] present a technique for detecting faults in Context-Aware Adaptive Applications (CAAAs) by defining a formal, Finite State Model of Adaptation (A-FSM) and then analyzing the model for finding adaptation faults. The A-FSM model represents the execution of a CAAA by explicitly connecting context updates with adaptations of the application and helps to isolate adaptation faults caused by erroneous rule predicates and asynchronous context updates.

In scenario-based testing, suitable techniques for obtaining scenarios are needed. An interesting approach may be based on “event-patterns”, a representation of peculiar event sequences that abstract meaningful test scenarios. An event-pattern may involve one or more contextual entities and possibly trigger a faulty behavior of the application. These patterns may be defined manually or on the basis of bug report analysis. Event-patterns are often used for rapid testing of embedded systems, too [108].

In the following section, possible approaches for using event-patterns for testing mobile applications are analyzed.

4.3 Techniques for Event-Patterns Based Testing

An event-pattern can be defined as a notable sequence of contextual events that may be used to exercise the application. It may be specified by a name, a textual description and the corresponding event sequence that must include one or more events. The sequence can be defined by appropriate regular expressions specifying optional, mandatory, iterative events and so on.

As an example, Table 4.1 lists some event-patterns that have been manually defined after a preliminary analysis conducted on the bug reports of open source applications available on public repositories like GitHub[2] and GoogleCode[3]. An event-pattern may be included in other event sequences, or used in isolation to test an app.

For automating test execution, each event-pattern can be associated with a test class that exposes and executes methods able to trigger the defined sequence events. As an example, Table 4.2 reports the method to be executed for three patterns defined using the Java language.

Once an event-pattern repository is available, it will be possible to generate test cases either manually or by semi-automatic approaches. In the following, three

[2]https://github.com
[3]http://code.google.com

Table 4.1: Some Event-Pattern Examples ('+' means *one or more*)

Event-Pattern Name	Event-Pattern Description.	Event-Pattern Specification
LRGPS	Loss and successive recovery of GPS signal while walking.	(locationChange)+, GPSLoss, GPSRecovered, (locationChange)+
NI	Network Instability.	(NetworkEnabled, NetworkDisabled)+
EGSW	The user Enables the GPS provider through the settings menu and starts a Walk.	openSettings, GPSOn, (locationChange)+
SIE	Arrival of a phone call when the device is in Stand-by. Then the call ends before the user accepts it.	standBy, IncomingPhoneCall, phoneCallEnds
UP	USB plugging in after any other event except the event itself.	USBUnplugged, USBPlugged
MSVC	Magnetic Sensor value changed after any other event except the event itself.	magneticSensorValueStatic, magneticSensorValueChange
IPC	Incoming of a phone call after any other event except the event itself.	ˆIncomingPhoneCall, IncomingPhoneCall

examples of testing techniques that exploit the event-patterns for testing a mobile app.

Manual technique (T1): A tester manually uses event-patterns to define scenario-based test cases that include one or more instances of event-patterns. The tester can add the needed assertions manually, or test cases can just check the occurrence of crashes. As an example, let suppose that the tester wants to test the app behavior in the following scenario. The user activates the GPS provider in the settings menu of the application, and begins to cross the path that goes from point A to point B through N points. At point X of the navigation, the application loses the GPS signal and recovers it at the point Y. This scenario includes instances of two event-patterns, EGSW and LRGPS respectively. The tester can reuse the

Table 4.2: Examples of Implementations of Event-Patterns

Pattern	Pattern Class Signature	Execute Public Method

EGSW	public class EGWS {
	...
	public void execute(Point A, Point B, ArrayList<Point>Route);
	private void OpenSettings();
	private void EnableGPS();
	private void Navigate(Point A, Point B, ArrayList<Point>Route);
	...
	}
		public void execute (Point A, Point B, ArrayList<Point>Route){
		OpenSettings();
		EnableGPS();
		Navigate(A,B,route);
		}

LRGPS	public class LRGPS {
	...
	public void execute(Point A, Point Y, Point B, ArrayList<Point>Route);
	private void GPSLocationChange(Point X);
	private void GPSLoss();
	private void GPSRRecovered();
	private void Navigate(Point A, Point B, ArrayList<Point>Route);
	...
	}
		public void execute (Point X, Point Y,Point B, ArrayList Route) {
		GPSLocationChange(X);
		GPSLoss();
		GPSRRecovered();
		GPSLocationChange(Y);
		Navigate(Y+1,B,Route);
		}

SIE	public Class SIE {
	...
	public void execute();
	private void deviceGoesInStandBy();
	private void incomingPhoneCall();
	private void phoneCallEnds();
	...
	}
		public void execute() {
		deviceGoesInStandBy();
		incomingPhoneCall();
		phoneCallEnds();
		}

code of these patterns to write the corresponding scenario test. Listing 4.1 shows a possible implementation of the scenario test reusing the pattern code.

Listing 4.1: Scenario Test Case

```
public void testScenario00038() {
  EGSW.execute(A,X-1,routeAX);
  LRGPS.execute(X,Y,B,routeYB);
}
```

Mutation-based technique (T2): Event-patterns are used to modify existing test cases, by applying mutation techniques that add event-pattern sequences inside already existing test cases (defined either manually, by Capture & Replay techniques or automatically). As an example, the tester may want to prove the absence of crashes in an application scenario where, after any user event, the device goes in stand-by and a phone call comes (pattern SIE). An approach that could be used for mutating the test cases is the one proposed by Barbosa et al. [22] for GUI testing where test cases made by sequences of events are altered automatically by introducing mutations in order to generate behaviors corresponding to errors that users typically make. In this case, the tester may automatically modify existing JUnit test cases, obtained using an Android GUI Ripper [12] [13] [9], by applying the event-pattern SIE, as it is shown in Table 4.3.

Table 4.3: An Example of Mutation of a Test Case by an Event-Pattern

Test Case before mutation	**Mutated test case**

public void testTrace00004() {
fireEvent (16908315, 16, "OK", "button", "click");
fireEvent (2131099651, 6, "","button", "click");
fireEvent (0, "", "null", "openMenu");
}
	public void testTrace00004_EP_SIE() {
	fireEvent (16908315, 16,"OK", "button", "click");
	SIE.execute();
	fireEvent (2131099651,6, "", "button", "click");
	SIE.execute();
	fireEvent (0, "","null", "openMenu");
	SIE.execute();
	}

Exploration-based technique (T3): Event-patterns are used during an automatic black-box testing processes that is based on dynamic analysis of the mobile application. In this case, an app exploration technique, like the one reported in Algorithm 1 in Chapter 3, may be used to define test cases and execute them at the same time.

Although the proposed techniques are applicable to any mobile applications, their feasibility is preliminary assessed in the context of Android mobile applications. In particular, to implement the proposed techniques two main technological problems related to the Android platform have to be solved, that are:

a) defining a solution for dynamically recognizing the context event classes which the app is able to sense and react at a given time;

b) defining techniques for triggering the context events.

The proposed solutions are presented in the following section.

4.4 Implementing Event-Based Testing in the Android Platform

As anticipated in Section 2.1, Android Applications are Event Driven System written in Java that run within an instance of a virtual machine. To solve the problem of dynamic recognition of the context event classes which the application is able to sense and react, different solutions can be adopted. Indeed, the set of context events that the application is able to sense and react to includes two distinct subsets. The former subset includes events that can be sensed by listeners and managed by the relative handlers defined by the running component itself. This set can be deduced by Java Reflection techniques, since Android applications usually dynamically declare listeners at run-time and code static analysis would not suffice. The latter subset includes events that may be managed by other app components and notified by means of Intent Messages. This set can be obtained by means of static analysis of the Android Manifest XML file of the application by searching for *intent-filter* tags reporting the set of Intent Messages to which any component of the application is sensible.

For triggering the context events, different techniques are developed.

A first solution exploits the APIs provided by the *java.lang.reflection* package, a set of classes designed for dynamic querying of Java class instances. Using these APIs is possible to directly access and execute event handlers methods related to event listeners instantiated at a given time. This technique can be adopted to trigger any event having a registered event listener, such as GUI events.

As to the problem of raising sensor-related context events, a solution that fires fake events instead of real sensor events is adopted. To this aim, the Android Sensor

Framework classes included in the *android.hardware package* (that is responsible for sensor management in Android) are replaced with an ad hoc modified version of this package that includes classes for generating fake events. This solution disables the sensibility of the app to real sensor events at all. A similar solution is implemented in the OpenIntents SensorSimulator project [4], too.

As to the emulation of location changes of the device the APIs of the *LocationManager* class provided by Android is exploited. This class allows the system location services to be accessed. The services are used to obtain periodic updates of the geographical location of the device, or to fire an Intent Message when the device enters in the proximity of a given geographical location. To emulate such notification events the *addTestProvider()* method of the *LocationManager* class is used; this method creates a mock location provider that programmatically emulates location changes.

The techniques presented above can be used to implement testing tools supporting the execution of the techniques shown in Section 4.3. In particular, they are implemented to develop a tool that implements the T3 technique.

4.5 A Case Study

An exploratory case study is conducted, with the purpose of assessing how the effectiveness of an event-based testing technique varies when context events, not only GUI events, are taken into account. In particular, an implementation of the automatic testing technique T3 proposed in Section 4.3 is analyzed.

For this aim, some real-world Android applications were considered and each of them was tested by the an implementation of the T3 technique. The first time, the planning strategy was configured to perform systematically only user events.

[4]https://github.com/openintents/sensorsimulator

The second time, patterns each one including a different type of context event was executed. Lastly, the testing effectiveness, in terms of code coverage, achieved by each of the two executions was compared.

To perform this experiment the Android Ripper tool presented in Chapter 3 has been exploited. In the first experiment, the Android Ripper was configured to trigger only user events; the second time another version of Android Ripper, called **Extended Ripper**, was used. The *Extended Ripper* is based on Algorithm 1 and features an extended *ExtractionCriterion* able to extract fireable context-related events and a customized *RunEvents* operation that is able to execute the extracted events such as location changes, enabling/disabling of GPS, changes in orientation, acceleration changes, reception of SMS messages and phone calls, shooting of photos with the camera. Both versions of Android Ripper are able to systematically explore the app under test by searching for crashes, to measure the obtained code coverage and to automatically generate jUnit test cases reproducing the explored executions.

In the case study five real Android applications were tested. They belong to the category of Mobile Apps and used context data of different types. They are all published on Google Play[5] and their source code is freely available. Table 4.4 reports a brief description of their features.

Each app was exercised from the same starting context both by the Android Ripper and by the enhanced version of Android Ripper. The resulting code coverage in terms of lines of code (LOCs) and methods has been measured: Table 4.5 shows the coverage values obtained.

[5]https://play.google.com/store

[6]https://play.google.com/store/apps/details?id=net.pierrox.mcompass

[7]https://play.google.com/store/apps/details?id=net.androgames.level

[8]https://play.google.com/store/apps/details?id=name.bagi.levente.pedometer

[9]https://play.google.com/store/apps/details?id=edu.nyu.cs.omnidroid.app

[10]https://play.google.com/store/apps/details?id=org.wordpress.android

Table 4.4: Characteristics of the Tested Apps

Application	**Description**
Marine Compass[6]	App showing a virtual marine compass providing the correct north direction on the basis of the data provided by the orientation sensor
Bubble Level[7]	App transforming the device in a virtual spirit level on the basis of the data obtained by the orientation sensor and by the accelerometer
Pedometer[8]	App showing a set of statistics regarding the walking of a person on the basis of accelerometer data
Omnidroid[9]	App for managing of personal actions and tasks that takes into account received/sent SMS messages and phone calls.
Wordpress for Android[10]	Client for managing Wordpress blogs, that can interact with the camera for the insertion of photos in a blog and with the GPS for the insertion of localization data

Table 4.5: Code Coverage Results

App	**LOC Coverage**		**Method Coverage**	
	Android Ripper	**Extended Ripper**	**Android Ripper**	**Extended Ripper**
MarineCompass	435 (92%)	463 (97%)	25 (86%)	27 (93%)
Bubble Level	371 (60%)	464 (75%)	75 (65%)	85 (74%)
Pedometer	528 (66%)	544 (67%)	160 (71%)	161 (72%)
Omnidroid	3409 (56%)	3480 (57%)	789 (58%)	813 (60%)
Wordpress	4505 (45%)	4599 (46%)	779 (53%)	784 (53%)

As the data show, the LOC code coverage achieved by the Extended Ripper grew of about 5% for Marine Compass and 15% for Bubble Level with respect to the Android Ripper one; the method coverage, instead, increased of 7% and 9% respectively. This difference could be attributed to the fact that a relevant part of app code implementing context event handling was covered just by the Extended Ripper.

As regards the Pedometer app, the Extended Ripper reached just a slight increase in coverage with respect to the Android Ripper (about 1% additional LOC and method coverage). This datum depended on the presence of just one context-related event handler in the app code, the one responsible for managing the acceleration change event.

For Omnidroid and Wordpress for Android the Extended Ripper reached a slightly higher coverage with respect to the one obtained using Android Ripper (just 1% more LOC coverage). In particular, in Omnidroid the Extended Ripper covered also the event handlers related to the management of incoming phone calls and SMS messages, while in Wordpress it covered the code related to the camera, as well as the location change event.

These results show that event-based testing effectiveness actually is improved thanks to the considered comprehensive set of context events. The more the app uses data from the context, the more the improvement becomes relevant. This datum preliminarily showed the utility of the proposed techniques.

CHAPTER 5

AGRippin: a novel search based testing technique for Android applications

In this chapter I present AGRippin (that is an acronym for **A**ndroid **G**enetic **Rippin**g) a search based testing technique applicable to Android applications with the purpose to generate test suites that are both effective in terms of coverage of the source code and efficient in terms of number of generated test cases. The proposed technique is based on the combination of genetic and hill climbing algorithms. A case study involving five open source Android applications is carried out to demonstrate that the technique is more effective than an Hill Climbing technique based on the systematic exploration of the GUI events executable on an Android application. The work described in this chapter is done in collaboration with the *REvERSE Research Group*[1].

5.1 Search Based Testing

Search Based Software Testing [3] is a specialization of Search Based Software Engineering (SBSE) [48] [31] related to the application of metaheuristic techniques to the problem of automatic generation of test cases optimizing the fault finding

[1]http://reverse.dieti.unina.it/

or the code coverage with a reasonable effort. A survey by Ali et al. [4] shows that there is a great interest in the research community for the application of metaheuristics techniques to problems related to automatic test case generation. In particular, in the set of metaheuristics algorithms, genetic algorithms are often used [4]. Genetic algorithms try to imitate the natural process of evolution: a population of candidate solutions, called chromosomes (i.e. test cases) is evolved using search operators such as selection, crossover, and mutation, gradually improving the fitness value of the individuals, until an optimal solution has been found or the search is stopped after a fixed time or a fixed number of evolutions.

At the time of writing, only a single contribution related to the application of Search Based techniques to mobile application GUI testing can be found in literature. Mahmood et al. [69] present a search based technique supported by the EvoDroid tool for evolutionary testing of Android applications. EvoDroid automatically extracts two static models of the application under test, i.e. the Interface Model and the Call Graph Model and generates evolutionary tests on the basis of these models.

5.2 The AGRippin Tecnique

According to the terminology of genetic algorithms, the *solution* proposed by the algorithm is, at each iteration, an evolved test suite that is composed of a *population* of *chromosomes* corresponding to test cases. Each chromosome is composed of *genes* corresponding to basic interactions with the application under test (AUT).

The effectiveness η of a test suite T can be defined as the fraction of lines of source code (LOCs in the following) of the AUT covered by at least one of the test cases composing the test suite generated by the algorithm. It can be evaluated by the following formula:

$$\eta(T) = 100 * \frac{|\bigcup_{t \in T} Cov(t)|}{|LOC|}$$

where $t \in T$ is a test case included in the test suite T, $Cov(t)$ is the set of lines of code that is covered by the test case t and LOC is the set of lines of code of the AUT.

The efficiency ϵ of a test suite T can be defined as the ratio between its effectiveness and the number of generated test cases:

$$\epsilon(T) = \frac{\eta(T)}{|T|}$$

The AGRippin technique adopts a constraint of genetic algorithms for which the size of the population is constant at each iteration and is equal to the size of the initial population. Due to this constraint, the test suite generated by the algorithm having the maximum effectiveness is also the one having the maximum efficiency.

In the next subsection are described the characteristics of the technique in terms of chromosome representation, metrics for fitness evaluation, techniques for crossover, mutation, selection, and combination with a hill climbing technique.

5.2.1 Representation

The test suites generated by the technique are composed of test cases that are sequences of interactions with the GUI of the AUT. The application GUIs are abstracted according to the conceptual model described in Subsection 3.2.2, shown in Figure 5.1.

5.2.2 Crossover

The crossover operator exploited in the implementation is a Single-Point Crossover. Given two Test-Cases t_1 and t_2, the operator randomly chooses two pairs $\{G_i, A_i\} \in t_1$ and $\{G_j, A_j\} \in t_2$ and operates the crossover operation as shown in Figure 5.2.

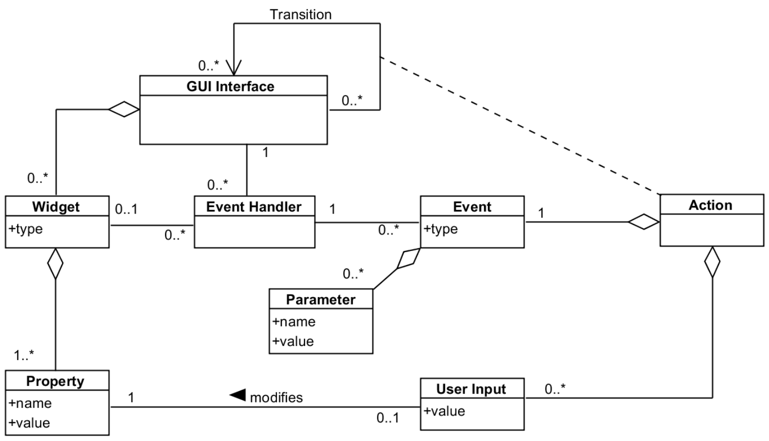


Figure 5.1: Conceptual Model of a GUI Interface

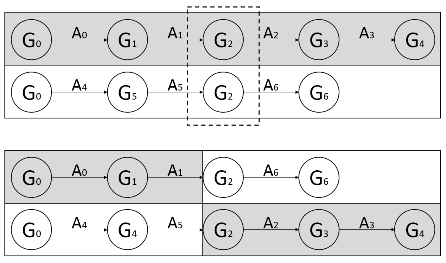


Figure 5.2: Crossover Example

A problem of this crossover operator is that it may generate sequences that do not correspond to executable test cases. If the generated test cases cannot be executed, they have to be discarded and the crossover operator has to be repeated until it generates a pair of executable test cases. In order to reduce the occurrence of such non-executable test cases, a technique is proposed to candidate pairs of test cases and cut points for which the crossover operator should be applicable, based of two heuristic criteria of equivalence between GUI interfaces and between actions. The two heuristic criteria of equivalence are defined in the following ways:

EC1 Two GUI interfaces are considered equivalent if they include the same set of widgets and they define the same set of event handlers.

EC2 Two actions are considered equivalent if they are associated to the same user actions and the same event.

Let's consider two test cases $t_1 = (G_0, ..., G_i, A_i, ...)$ and $t_2 = (G_0, ..., G_j, A_j, ...)$ having the same starting GUI interface G_0. The pairs (G_i, A_i) and (G_j, A_j) are a candidate crossover point for our heuristic technique if they satisfy all these four criteria:

C1 the two GUI interfaces G_i and G_j are equivalent according to the $EC1$ criterion;

C2 the two actions A_i and A_j are not equivalent according to the $EC2$ criterion;

C3 the subsequence of t_1 which precedes the GUI interface G_i and the subsequence of t_2 which precedes the GUI interface G_j are not composed of a sequence of GUI interfaces and actions that are all respectively equivalent (according to the two criteria $EC1$ and $EC2$), *and* they are not both empty;

C4 the subsequence of t_1 which follows the action A_i and the subsequence of t_2 which follows the action A_j are not composed of a sequence of GUI interfaces and actions that are all respectively equivalent (according to the two criteria $EC1$ and $EC2$), *and* they are not both empty.

It's interesting to note that the first criterion avoids to select crossover points for which the action A_j is not applicable to the GUI interface G_i or the action A_i is not applicable to the GUI interface G_j. The other three criteria avoids the selection of crossover points that generate two test cases that are too similar or identical to the original ones.

As an example, let's observe the crossover example in Figure 5.2, in which equivalent GUI interfaces are labeled with the same label. We can verify that the selected crossover point (corresponding to the pairs (G_2, A_2) and (G_2, A_6)) is the unique one that satisfies all the four criteria (the pairs (G_0, A_0) and (G_0, A_4) satisfy the first two and the fourth criterion but they do not satisfy the third criterion because they are both preceded by an empty sequence).

The crossover points are randomly chosen in the set of the ones that satisfy these criteria. The test cases t_1 and t_2 are not removed from the test suite after the execution of the crossover operator, in concordance with the techniques proposed in the steady state genetic algorithms [98] (for example the Genitor one proposed by Whitley [116]). These techniques cause an increase in the population size that is restored to its initial size by the selection operator that is presented in the following. The crossover operator may be executed multiple times in the same iteration of the algorithm. We define the *crossover ratio* as the ratio between the number of test cases generated by the crossover at each iteration and the number of test cases of the initial solution.

5.2.3 Mutation

The mutation operator proposed in this technique modifies the Actions by mutating the values of the User Inputs or the Event Parameters values. In each mutation, the value of a single parameter of the action is changed to a new value belonging to a static set of equivalence classes according to the parameter type. As an example, the value of an editable text field may be set to a random string, a number or a correct email address while the location parameter of a GPS event may be set to coordinates values over or under the equator. The new test obtained after a mutation could not be executable if the GUI interface reached after the execution of the mutated action is not equivalent to the one reached by the original action. In this case, we consider that the new mutated test case terminates with the mutated action and the new test case is shorter than the original one.

The mutation operator randomly selects the test case and the action to be mutated in all the test suite. Mutated test cases are added to the test suite and the original ones are not removed. The *mutation ratio* is defined as the ratio between the number of test cases generated by the mutation operator and the number of test cases of the initial solution.

5.2.4 Fitness Evaluation

Two distinct Fitness measures are defined: the Global Fitness (that is the effectiveness η of the generated test suite and is measured in terms of the code coverage reached by the test cases of the test suite as described above in this section), and a Local Fitness expressing the degree of diversity of a single test case with respect to the set of the test cases of the test suite.

The Local Fitness measures ranks the individuals in terms of their potential contribution to the Global Fitness of the solution and of their diversity. To this aim, a *rank* measure that is able to order all the test cases of the test suite is proposed. The Local Fitness measure is composed of two components named $F1$ and $F2$. The first component $F1$ may assume the following three values, in order of decreasing rank:

- L_1, if the test case covers one or more lines that are not covered by any other test case;
- L_2, if the test case has a coverage set that (i) includes only lines of code that are covered by at least another test case of the test suite but that (ii) is not included in the set of lines covered by any other test case of the current test suite;
- L_3, if the test case has a coverage set that is included in the coverage set of at least another test case of the solution.

As regards the set of test cases having the same coverage set, the algorithm conventionally assigns a L_2 value to one test case (randomly selected) of the set and the L_3 value to all the other test cases of the set. Intuitively, test cases having a L_1 value are the ones that should be preserved to avoid a sure loss of effectiveness, whereas test cases having a L_3 value are the better candidates to be filtered out by the selection operator.

The second component $F2$ of the Local Fitness represents a weighted measure of code coverage and is defined by the following formula:

$$F2(t) = \textstyle\sum_{l \in Cov(t)} w(l)$$

where:

- $Cov(t)$ is the set of lines of code that are covered by the test case t
- $w(l)$ represents the relative weight of the coverage of the line l. It is defined as:

$$w(l) = \frac{1}{\sum_{u \in T} c(u)}$$

where $c(u) \in \{0, 1\}$. It is 0 if $l \notin Cov(u)$, 1 elsewhere.

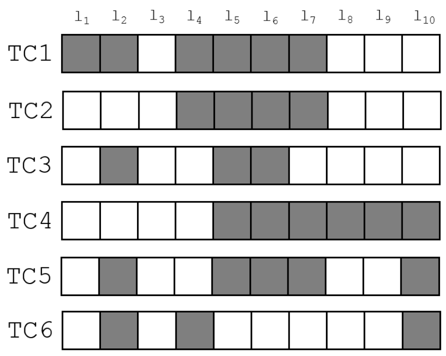


Figure 5.3: Test-Case Fitness Evaluation

$F2$ gives a measure of the relative importance of the coverage provided by a test cases in the context of a test suite because the coverage of lines that are covered by few test cases has a higher weight than the coverage of lines covered by many test cases. $F2$ is used to order test cases having the same $F1$ values.

As an example, let's consider the test suite shown in Figure 5.3 in which there is an AUT composed of 10 LOCs labeled as $\{l_1, ..., l_{10}\}$ and a Test Suite T = $\{TC1, ..., TC6\}$. The coverage of each test case is depicted in Figure 5.3 where black boxes corresponds to covered lines whereas white boxes corresponds to uncovered lines.

In order to evaluate the Local Fitness $F1(TC1) = L_1$ and $F1(TC4) = L_1$ have been assigned because they are respectively the unique test cases covering the line l_1 and the two lines l_8 and l_9. The values of $F1(TC2)$ and $F1(TC3)$ are instead set to L_3 because their coverage sets are respectively included in the ones of $TC1$ and $TC5$. The $F1$ value of the remaining test cases (i.e. $TC5$ and $TC6$) is set to L_2. In order to evaluate the $F2$ values for each test case, the weights w of each line l have to be evaluated. For example, the weight of line l_1 is 1 because it is covered exactly by one test case, whereas $w(l_2) = \frac{1}{4}$ because the line l_2 is covered by four test cases and so on. The Fitness Function of the test case $TC1$ is then equal to:

$$F2(TC1) = w(l_1)+w(l_2)+w(l_4)+w(l_5)+w(l_6)+w(l_7) = \tfrac{1}{1}+\tfrac{1}{4}+\tfrac{1}{3}+\tfrac{1}{5}+\tfrac{1}{5}+\tfrac{1}{4} = 2.23$$

The $F2$ values for each test case are reported in Table 5.1. In this table the test cases are ordered for decreasing values of $F1$ and, for test cases with the same $F1$ value, for decreasing values of $F2$. The $RANK$ column expresses the ordering position between all the test cases of the test suite.

RANK	**t**	$F1$	$F2$
1	TC4	L_1	2.98
2	TC1	L_1	2.23
3	TC5	L_2	1.23
4	TC6	L_2	0.91
5	TC2	L_3	0.98
6	TC3	L_3	0.65

Table 5.1: Test-Case Classification Example

5.2.5 Selection

The selection operator restores the size of the test suite to its initial value (corresponding to the size of the initial test suite) by deleting the test cases having the worst values of Local Fitness in concordance with the rank selection operator firstly proposed by Baker [19].

The fraction of test cases that are selected for deletion at each iteration is named *turnover ratio* and it is the sum of the *crossover ratio* and of the *mutation ratio*. As an example, if the crossover ratio is 1/3, then the test cases TC2 and TC3 shown in Table 5.1 have to be deleted.

5.2.6 Combination Technique

By means of the application of the crossover and of the mutation operator GUI interfaces that are not equivalent to any of the already visited ones may be discovered. These new GUI interfaces contains different sets of widgets and event handlers with respect to the other ones. This represents a positive achievement in terms of global fitness because new code corresponding to the execution of these event handlers may be executed.

In the AGRippin technique is proposed a *combination* of the genetic technique with an Active Learning technique that will be started only when a new GUI interface is discovered. This technique aims at the systematic generation of new test cases including at least an event of each new discovered GUI interface and is very similar to the one we have proposed in the past [12]. This technique can be seen as a Hill Climbing technique because it selects at each iteration the most promising sequences, i.e. the ones in which at least a new line, corresponding to a new event handler call is covered. In order to restore the size of the test suite to its initial value, a re-execution of the selection operator has to be carried out after each execution of the Active Learning technique.

The adoption of hybrid algorithms combining genetic and hill climbing algorithms have been already presented in literature, with good results [86] and some criticism. We adopted this solution for two reasons: (1) because our specific implementation of the mutation operator is not able to generate new events but only to mutate their parameters and (2) to accelerate the process of exploring the interactions related to portions of the application that are discovered but not explored by crossover and mutation operators.

5.3 Case Study

This section reports the results of some case studies that carried out with the aim to assess the effectiveness of the proposed search based testing technique. The technique was implemented in the context of Android applications and is applied to five open-source Android applications.

The test suites generated by the technique are compared with the ones generated by the Android Ripper Tool [12] that was developed in the past. It realizes a Model Learning technique for the exploration of the GUI of Android applications. Since the Android Ripper explores at each iteration the GUIs of an Android application by executing an event that have never been executed before, this technique can be considered as a kind of Hill Climbing technique because an increment in code coverage is surely expected by the execution of each new event.

The purpose of the experimentation is to provide an answer for the following research question:

RQ: Are the test suites generated by the proposed technique more effective than the ones generated by the considered Hill Climbing technique?

The effectiveness of the generated test suites is measured (in percentage) as the fraction of lines of code of the AUT that are covered at least once by at least a test case of the test suite T:

$$\eta(T) = 100 * \frac{|\bigcup_{t \in T} Cov(t)|}{|LOC|}$$

5.3.1 Subjects

Five real-world open source Android Applications have been selected for the study; they are all published and freely available on the Google Play market. Some details about these application are reported in Table 5.2. They are all medium sized applications, with a number of LOCs varying from 2308 lines (AUT1) to 6770 lines (AUT3).

Table 5.2: Android Applications (AUTs)

	Application	Description	Link	LOCs	Activities
AUT1	AardDict 1.4.1	A dictionary application	https://github.com/aarddict/android	2308	7
AUT2	TomDroid 0.7.1	A manager for notes	https://code.launchpad.net/tomdroid	4167	10
AUT3	OmniDroid 0.2.1	A manager for device automated tasks and actions	https://code.google.com/p/omnidroid/	6770	16
AUT4	AlarmClock 1.7	An alarm clock	https://code.google.com/p/kraigsandroid/	2320	5
AUT5	BookWorm 1.0.18	A manager for book collections	https://code.google.com/p/and-bookworm/	3190	10

Application preconditions can affect the effectiveness of the generated test cases [12]. For the purpose of this experimentation, the same set of preconditions were chosen for each application and used in all the experiments with both the techniques (as an example, for AUT1 the same dictionary in the SD card before the execution of each test case has been preloaded).

5.3.2 Experiment Environment and Setup

The experimentation was carried out by using two tools: the Android Ripper tool and the AGRippin tool.

The Android Ripper tool[2] [12] was used to systematically explore the GUIs of Android applications with a breadth-first strategy. Each branch of the exploration carried out by the Android Ripper tool is terminated when a GUI interface is found that is equivalent (in the sense that it has the same widgets and event handlers) to a previously visited one. The Android Ripper tool produces a test suite composed of test cases corresponding to the explored execution paths.

The Android Ripper tool is composed of two main components. The *Driver* component is responsible for the execution of the exploration algorithm and for the generation of the resulting test cases in form of Android JUnit test cases exploiting

[2]https://github.com/reverse-unina/AndroidRipper

the Robotium library[3]. The *Device* component is deployed and executed in the context of an Android emulator and is able to execute actions on the AUT, to extract the obtained GUI interfaces and to send their description to the *Driver* component via the Android Debug Bridge (ADB)[4] utility. The code coverage was measured by means of the Emma utility[5] included in the Android SDK.

The AGRippin tool was realized on top of the Android Ripper tool by implementing an *AGR* component responsible of the execution of the proposed technique. The *AGR* component interacts with both the components of the Android Ripper tool.

The experimentation was carried out on 6 different Intel I5 PCs with a clock frequency of 3.0GHz, 4GB of RAM and Windows 7 64bit operating system. On these machines we installed an Android Virtual Device [6] (AVD) emulating the Android Gingerbread 2.3.3 operating system, with 512MB of RAM and an emulated 64MB SD Card.

The experimentation was started by carrying out an exploration of each AUT by means of the Android Ripper tool that generated a test suite. These test suites were considered as a result of an Hill Climbing exploration because the Ripper strategy consists of the selection, at each step, of the most promising action, i.e. of an action that has not been previously executed. The test suite generated by the Android Ripper tool has been used, too, as the initial solution of the search based testing technique.

The AGRippin technique was configured in our experimentation by fixing the parameters shown in Table 5.3. The *Crossover ratio* and the *Mutation ratio* respectively represent the fraction of the test cases of a test suite that are involved in a

[3]https://code.google.com/p/robotium/
[4]http://developer.android.com/tools/help/adb.html
[5]http://emma.sourceforge.net/
[6]https://developer.android.com/tools/devices/index.html

crossover or in a mutation at a given iteration. In accordance with the suggestions of Mitchell et al. [83] we set a higher value for the Crossover ratio with respect to the Mutation ratio. The *Number of Iterations* represents the termination condition of our algorithm in terms of the number of performed iterations. We fixed an arbitrary value of 30 in this experimentation. In order to take into account the randomness of our search based testing technique, the AGRippin technique was executed six times with six different seeds for any AUT.

Parameter	**Value**
Crossover ratio	20%
Mutation ratio	5%
Number of Iterations	30

Table 5.3: Configuration Parameters Values

5.3.3 Results and Discussions

Table 5.4 reports the results obtained by the execution of our experimentation on the five AUTs.

The first column of the table reports the effectiveness $\eta(T_0)$ of the test suite T_0 generated by the Hill Climbing (HC) technique implemented by the Android Ripper tool. The columns labeled $\mu(\eta(T))$ and $\sigma(\eta(T))$ respectively report the average and the standard deviation of the effectiveness η of the test suites T generated by the AGRippin technique (abbreviated in AGR in Table 5.4) in six different executions featuring different random seeds. The fourth column of the table reports the maximum number of new interfaces discovered by AGRippin that were not discovered by HC. The fifth column reports the number of test cases composing both the test suites generated by HC and AGRippin. Finally, the last two columns respectively report the HC execution time and the average execution

Table 5.4: Experimental Results

	HC	**AGR**				**Execution Time (hours)**	
	$\eta(T_0)$	$\mu(\eta(T))$	$\sigma(\eta(T))$	**New Interf.**	$\lvert T\rvert$	**HC**	**AGR**
AUT1	43.07%	67.10%	0.26%	1	51	3.5	22
AUT2	28.08%	32.61%	2.45%	0	51	2.5	30
AUT3	51.58%	58.31%	2.28%	0	162	6	55
AUT4	66.90%	68.00%	1.21%	0	68	2.8	20
AUT5	40.34%	47.22%	0.45%	1	50	3.2	23

time of AGRippin (after 30 iterations), measured on the same machines.

The results in this table show that for all the considered AUTs the AGRippin technique is able to provide an increase in coverage with respect to the Hill Climbing technique that varies from 1% (for AUT4) to 24% (for AUT1), so can be concluded that the proposed RQ has a positive answer. As regards the execution time, the average time needed to execute 30 iterations with AGRippin varies from 6 to 12 times the amount of time needed to execute HC. The values of standard deviation $\sigma(T)$ show that the effectiveness of AGRippin depends on the randomness in a remarkable way. Can be hypothesized that the execution of a larger number of parallel sessions can provide improvements in the effectiveness of the AGRippin technique without increasing the execution time.

In order to show an example of the dependence of the effectiveness on the number of iterations, Figure 5.4 reports the effectiveness trends observed for the Bookworm application (AUT5). The figure shows the trends of the coverage of the test suites obtained by six different executions (with six different random seeds) of the AGRippin technique (named AGR1, AGR2, AGR3, AGR4, AGR5, AGR6 in figure) as the number of iterations increases. The dashed line shown in figure represents the coverage percentage provided by the test suite generated by the HC technique. Each execution of AGRippin constantly reaches higher values of effectiveness than HC. In the Bookworm application, a new interface has been discovered by each AGRippin execution and in these cases a large portion of new code have been systematically been explored by the Hill Climbing technique. Its effect can be noted by observing the rapid rising in the effectiveness that occurs once for each AGRippin execution. The values of effectiveness after 30 iterations are slightly different between them. On the basis of this phenomenon can be hypothesized that the execution of a larger number of iterations may provide better results and a reduced dependency on randomness. Similar considerations can be done for all the other AUTs.

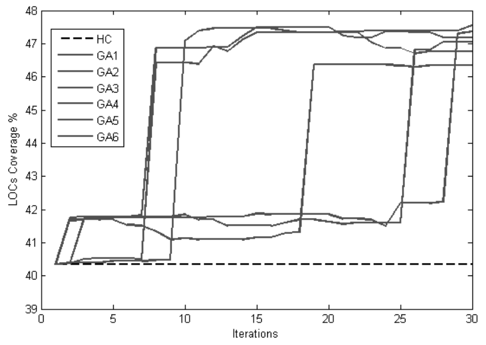


Figure 5.4: Effectiveness Trends for AUT5

In order to provide more details about the capability of the AGRippin technique to cover new portions of code, the differences in coverage between the executions of the HC and AGRippin techniques was examined in details. Improvements in coverage due to the Crossover operator, to the Mutation operator and to the Combination technique were recognized.

As regards the improvements due to the crossover operator, in some cases the mixing of two different test cases produced new execution sequences that are able to show different behaviors of the AUT. As an example, for AUT2 the crossover operator generates a new test case executing the backup functionality after the changing of its settings causing the execution of a different backup scenario. Another example of new code revealed by a crossover operator is, in AUT5, the one related to the sequential execution of the insertion of a book in the book list followed by the visualization of this book list. In the test suite generated by HC, the visualization was executed only with an empty book list whereas AGRippin were able to visualize book lists that are not empty.

As regards the improvements in effectiveness due to the mutation operator, two exemplar cases are reported in the following. An AUT1 functionality is the search in a vocabulary of a string included in an input text field. Whereas the HC technique only provided a random text to this input field, the mutation operator implemented in AGRippin generated a new test case with an input text value belonging to the equivalence class of the English words. This mutation caused the execution of a portion of code related to the retrieval of the word in the dictionary and to the visualization of a new interface showing the list including one or more search results. Another example is the one found in AUT4, where the insertion of an input value belonging to the negative numbers equivalence class in a specific text field caused the execution of a portion of code executing a validating check that was not tested by HC.

Finally, the combination technique was applied in two cases regarding the new interfaces discovered in AUT1 and AUT5 (corresponding to two of the cases described above). In these cases the application of the HC technique on these new interfaces caused a great improvement in code coverage (about 5% of improvement in both the cases).

CHAPTER 6

Exploiting the saturation effect in automatic random testing of android applications

In this chapter I describe a possible solution to the problem of stopping a Random testing process at a cost-effective point, where test adequacy is maximized and no testing effort is wasted. A fully automatic Monkey Fuzz Testing (MFT) process that is able to find this point by exploiting the *saturation effect* and the *predictability property* of random testing techniques is presented. The validity of the approach in finding saturation points is then shown by an experiment where 18 real Android applications are tested. The work described in this chapter is done in collaboration with the *REvERSE Research Group*[1].

6.1 Introduction

Random testing is a black-box software testing technique where programs are tested by generating random, independent inputs. Nevertheless in the past random testing was considered less promising than systematic testing techniques [87], many other works in literature empirically demonstrated their effectiveness in many different contexts [92], [46], [44], [115], [28]. Arcuri et al. [16] addressed random testing from a theoretical point of view by proposing a mathematical model for

[1]http://reverse.dieti.unina.it/

describing the effectiveness of random testing and comparing it against partition testing. Moreover they presented some novel theoretical results regarding effectiveness, scalability and predictability of random testing. This work focused on testing techniques that randomly choose input values from the input domain, while our work focuses on techniques that choose events from the event domain.

Over the last several decades, variants of random testing have gained popularity in automated quality assurance testing for both conventional software applications as well as graphical user interface (GUI) frontends. These random testing techniques have several benefits over other testing methods: they are fully automatic, inexpensive, relatively easy to use, and surprisingly effective at finding bugs [34][37][103]. For example, Miller et al. [80] used random testing to reveal a wealth of command-line faults within Unix utilities; similarly, Miller et al. [79] used random testing to reveal faults in 10 out of 135 command-line utilities and 22 out of 30 GUI-based utilities within MacOS applicationsṪhe general validity and importance of random testing has been further evaluated and supported in a seminal work by Duran & Ntafos amongst others [34][37][103].

The growing use of random testing is particularly evident in the mobile app realm, where many platforms and developers have adopted *Monkey Fuzz Testing* (MFT), a technique that sends random button presses and mouse events to an app[2].

Although tools for MFT are growing in availability and popularity, there are still open issues regarding their general use. One particular issue is how a tester may determine when the testing process should be stopped. For example, MFT tools are commonly used to implement simple testing processes where the tool is run on an old, slow computer of the testing lab and a tester periodically checks its progress [93]. Although this approach is reasonable, it is difficult for the tester to

[2]e.g., see monkeyfuzz.codeplex.com and developer.android.com/tools/help/monkey.html

know when the testing process has reached a point where no further code coverage or fault detection can be achieved. As a result, the tester has no other option but to make an educated guess regarding two choices: stop testing or allow the process to run until some later termination criteria are met. The first choice may stop the program prematurely resulting in untested code, and the latter may stop the process later than needed and result in wasted time and computing resources. Both cases are non-optimal and rely on some expertise of the tester to use a proper termination criterion and point.

Many testers let the test process run until a certain amount of time has elapsed without the process discovering new faults. While this solution is functional, it is hardly optimal.

Several studies in literature studied the predictability of the performance of random testing techniques. Ciupa et al. [29] [30] explored the predictability of random testing in terms of fault detection capability in the context of object-oriented programs written in Eiffel. More recently, Furia et al. [38] have searched for a law able to relate the number of executed random test cases and the number of found faults and failures in Eiffel and Java applications. Sherman et al. [104] exploited the existence of a saturation effect occurring when the increase in coverage over a window of test runs (of a given size) is less than a fixed threshold and used this effect to define new adequacy criteria in concurrency testing.

In this chapter, a testing process is presented that tries to relieve responsibility from the tester by automatically determining a termination point based on the exploitation of the *saturation effect*, a well-known phenomena where the rate of convergence of a test case towards a specific test adequacy criterion decreases as the amount of test execution increases [67]. The result is that the testing process may stop finding faults in the program under test. The point at which no more progress is made towards achieving a test adequacy criterion is called *Saturation*

Point and it is also the optimal termination point for the program during random testing.

Determining the saturation point of a program can be elusive and is dependent on the preconditions of the app and the configuration settings of the tool chosen by the tester. For instance, in the Android platform the capacity of a GUI Ripper to discover faults and to cover the app source code sensibly depends on several preconditions including the types of events fired on the GUI, the timing between consecutive events, or input values provided to the input fields of the GUI [11]. Such choices are dependent on both the tester and the MFT tool the tester chooses to use. Therefore, in order to automatically determine a saturation point of the program, a specific preconditions of the app and a configuration of the tool are provided as an input, until the saturation effect is detected.

More specifically, the saturation point is determined by a process based on the simultaneous execution of several random testing sessions. By exploiting the predictability property of random testing shown by Arcuri et al. [16], the difference in code coverage between the active sessions is periodically assessed and the testing process is stopped when this difference is below a chosen critical threshold.

The fully automatic technique is implemented by means of an infrastructure including a MFT tool targeting the Android platform and a simulation environment to run each test. To validate the implemented approach, 18 Android apps have been selected from the Google Play Store and tested each with the testing infrastructure. The study showed that the termination points of all processes were also saturation points.

6.2 Monkey Fuzz Testing Tools

Monkey Fuzz Testing tools were first developed as a method of stress testing for both conventional software applications and those applications with a GUI front-end. They perform such activity by sending sequences of random keyboard or mouse events to the subject applications, with the aim of discovering crashes or other inconsistencies in the behavior of the applications.

MFT tools for mobile applications differ slightly from those for desktop GUI-based applications because they can be configured to send both user and system events.

Traditionally, both mobile and desktop MFT tools offer options which can be configured to implement different behaviors. E.g., it is often possible to set the delay between consecutive triggered events, the types of events to fire (such as click, tap, or other), how often an event gets triggered, the number of events to fire and so on. Before testing an application, test engineers are required to configure such options and choose the preconditions of the application under test, that is the state of the application under test before the test run. Once the tool is configured, it starts generating random events and sending them to the AUT that will reach a new state S. The process stops when a termination criterion is satisfied. This behavior can be described by Algorithm 1 presented in Chapter 3.

The $TerminationCriterion$ parameter may be based on aspects of the process, such as the number of events that have been fired or the amount of time spent testing, or it may be based on some adequacy measurement that determines whether sufficient testing has been executed. For instance, when using the statement coverage criterion as an adequacy measurement, the testing process can be stopped if all the statements have been executed, or the percentage of executed statements is greater than a given threshold [125].

The choice of the *TerminationCriterion* is always a relevant problem with automatic testing processes, since it is able to affect their effectiveness and cost. This stop condition is even more important with random testing techniques, which are notoriously affected by problems of reliability and efficiency. The reliability problems of random testing depend on the randomness of this technique: if the same tool is launched multiple times, even from the same initial preconditions, the testing results may sensibly differ, due to the randomness of the sequence of events sent to the application.

On the other hand, the inefficiency problems of random testing depend on the risk of wasting excessive testing effort in trying to achieve non-reachable test adequacy levels. It is well-known indeed that, any testing method is affected by a *Saturation Effect*, that is the tendency of the method of limiting its ability to expose faults in a program under test [58]. After reaching this limit, continuing testing the same method may cause significant waste of testing efforts. This limit is also called a *Saturation Point* in the literature and several authors tried to exploit it to define a termination point of testing [104].

These two problems are well illustrated by Fig. 6.1, which reports the code coverage percentage (as the number of sent events grows) of an example application that was achieved by two different testing runs of the same MFT tool. In each run the same tool configuration are used and the application is started from the same application preconditions, but used different seeds for generating different sequences of random events.

As the figure shows, there is an initial unreliability zone (instability phase) of the process (between 0 and about 3,000 events) where the testing runs achieve different and floating coverage results. The code coverage achieved by the former run (represented by the continuous line in the figure) is indeed initially lower than the latter's ones (represented by the dashed lines), but there is a trend inversion after about 500 events. On the other hand, after about 4,700 fired events, the coverage degrees of the two runs seems to converge and to reach a zone showing a possible saturation effect.

This code coverage trend yields to an important consequence. If the tester stopped the tool after less than 4,000 events, the test adequacy would be different depending on the considered run. Vice-versa, after firing more than 4,000 events, the coverage results of different testing runs tend to be similar and independent of chance. However, if the tester continues and fires more than 4,700 events, they will do nothing but waste both time and computing resources due to the saturation effect. Ideally, a tester would be able to identify the number of events large enough to overcome this instability zone and small enough to avoid entering the inefficiency zone. The tester could then stop the process at a point that represents an optimal trade-off between effectiveness and cost. Since beyond this point there is no an improvement of the testing adequacy, one could consider this point the saturation point of the application according to the definition given by Sherman et al. [104].

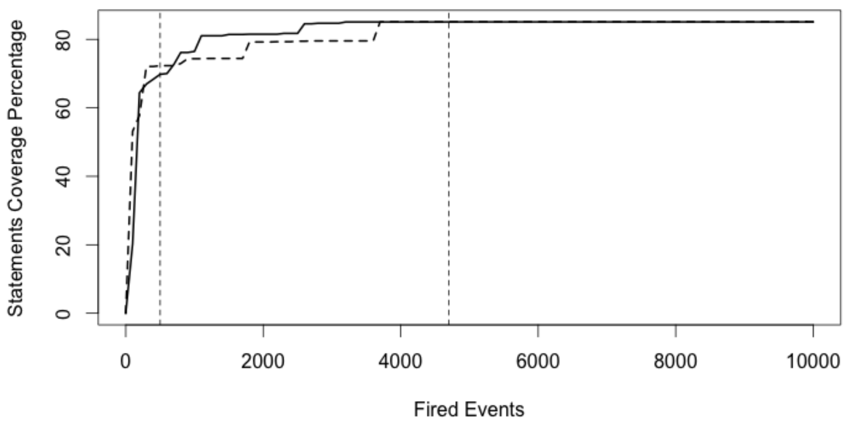


Figure 6.1: Code coverage of two testing runs of an MFT tool

6.3 The Testing Process

In this section, is presented a testing process based on MFT techniques that automatically stops at a potential saturation point of the process, defined as the point in testing where additional fired events result in no improvement in test adequacy. In order to reach this objective, our implementation utilizes a pool of random testing *sessions*. All sessions in the pool are given the same initial conditions, but are fed a different seed and sequence of consecutive random events to fire on the AUT. Exploiting the predictability property of random testing shown by Arcuri and Briand [16], one can accurately infer that there is a point P of this process where the difference in code coverage achieved by all sessions is equal to zero. This point of least difference may represent a saturation point of testing, indicating that all sessions reached the same test adequacy. On the basis of this property, such a point can be the optimal termination point for testing technique for the AUT. Of course, this calculated difference is dependent of the chosen pool of test sessions. The smaller the number of considered sessions, the greater the probability their code coverage be coincident at a *premature* saturation point. Vice-versa, the greater the number of sessions, the greater the likelihood they converge at an *authentic* saturation point. In the sub-sections that follow, the test process implementation is presented in greater detail.

6.3.1 The Testing Process Implementation

The testing process description requires the following definitions:

- AUT: it is the application under test.
- $AUTPr$: it is the set of AUT preconditions.

- *MFT*: it is a Monkey Fuzz Testing tool configured according to the *MFTSet* settings.
- S_i: it is a testing **S**ession of the MFT tool that sends a sequence of consecutive random events to the AUT.
- $S = \{S_1, \ldots, S_k\}$: it is a set of $k > 1$ testing sessions S_i. All the sessions start from the same initial state of the AUT but have different seeds.
- *SST*: it is the **S**et of executable **ST**atements composing the source code of the AUT.
- *SCS*(S_i, n): it is the **S**et of **C**overed **S**tatements of the application under test, after n events fired in S_i.
- *CSP*(S_i, n) it is the **C**overed **S**tatements **P**ercentage achieved by the session S_i after n random fired events. It is expressed by the following formula:

$$CSP(S_i, n) = \frac{\mid SCS(S_i, n) \mid}{\mid SST \mid} \times 100 \tag{6.1}$$

- *CSSC*(S, n): it is the **C**umulative **S**et of **S**tatements **C**overed by the testing sessions belonging to S after n fired events. It is defined by equation (6.2):

$$CSSC(S, n) = \bigcup_{i=1}^{k} (SCS(S_i, n)) \tag{6.2}$$

- *CCSP*(S, n): it is the **C**umulative **C**overage **S**tatement **P**ercentage reached by the testing sessions belonging to S after n fired events. It is defined by equation (6.3):

$$CCSP(S, n) = \frac{\mid CSSC(S, n) \mid}{\mid SST \mid} \times 100 \tag{6.3}$$

- *TerCond*(S, n): It is a predicate that is true after n events, if each session of S reached a statement coverage percentage that is equal to the cumulative one. In other words the predicate is true when all the sessions have actually covered the same statements of the AUT. It is evaluated by means of equation (6.4).

$$\begin{gathered} \mathit{TerCond}(S, n) = \mathit{TRUE} \iff \\ \mathit{CSP}(S_i, n) == \mathit{CCSP}(S, n)\ \forall\ S_i\ \in S \end{gathered} \tag{6.4}$$

- *TerP*(*TP*): it is the **Ter**mination **P**oint of the testing process *TP* representing the minimum number of events at which the termination condition is verified.

- *TerL*(*TP*): it is the **Ter**mination **L**evel indicating the cumulative coverage statement percentage reached by the testing process *TP* up to *TerP*.

Algorithm 2 Testing Process Algorithm

```
1: procedure TestingProcessExecution($AUT$, $AUTPr$, $MFT$, $MFTSet$, $k$)
2:     $S[] \leftarrow initSessions(AUT, AUTPr, k)$;
3:     $terCon \leftarrow FALSE$;
4:     $samplingStep \leftarrow STEP$;
5:     $fe \leftarrow 0$;
6:     while ($!terCon$) do
7:         $fireNextEvent(AUT, MFT, MFTSet, S[])$;
8:         $fe{+}{+}$;
9:         if ($fe == samplingStep$) then
10:            $SCS[] \leftarrow evalCov(S[])$;
11:            $CSSC \leftarrow evalCCov(SCS[])$;
12:            $terCon \leftarrow evalTerCon(CSSC, SCS[])$;
13:            $samplingStep \leftarrow samplingStep + STEP$;
14:    $stopSessions(S[])$;
15:    $terP \leftarrow fe$;
16:    $terL \leftarrow evalPercCov(AUT, CSSC)$
```

The Testing Process *TP* is a quintuple (AUT, *AUTPr*, *MFT*, *MFTSet*, k). The process is iterative and requires the periodic monitoring of the statement coverage percentages of k random sessions with a predefined sampling step. It is described by the pseudo code in Fig. 2. At each iteration, each session S_i had fired a number fe of events. The algorithm relies on the variables and method's invocations described below.

6.3.1.1 Constants and Variables

- *terCon*: it is a boolean variable assuming the value of the *TerCond*() predicate.
- *samplingStep*: it is an integer variable > 0, representing the sampling step, i.e., the number of fired events after which the termination condition will be evaluated.
- *STEP*: it is a integer constant > 0 defining the sampling period of the algorithm.
- *fe*: it is an integer variable representing the number of events that have been fired by all the sessions at a given iteration of the algorithm.
- k: it is an integer representing the number of testing sessions executed by the testing process.
- *S[]*: it is an array of k testing sessions.
- *SCS[]*: it is an array of statement sets. The i_{th} element of this array represents the set of statements that have been covered by the i_{th} random testing session after *fe* fired events.
- *CSSC*: it is the set of statements that have been covered by all the sessions after *fe* fired events.
- *terP*: it is an integer variable related to the termination point.

- *terL*: it is a double variable representing the value of the termination level.

6.3.1.2 Methods

- *initSessions(AUT, $AUTPr$, k)*: it launches the execution of k instances of the AUT from the same preconditions $AUTPr$, and starts k testing sessions belonging to the array $S[]$.
- *fireNextEvent(AUT, MFT, $MFTSet$, $S[]$)*: in each testing session of $S[]$, the *MFT* sends a random event to the AUT according to its settings *MFTSet*.
- *evalCov($S[]$)*: it evaluates the set of statements that are covered by each session of $S[]$.
- *evalCCov($SCS[]$)*: it evaluates the cumulative set of statements $SCS[]$ that have been covered by all the testing sessions.
- *evalTerCon($CSSC$,$SCS[]$)*: it computes the covered statement percentage reached by each testing session and the cumulative coverage percentage. Then, it evaluates the predicate described by equation (6.4).
- *stopSessions($S[]$)*: it stops the execution of the testing sessions belonging to $S[]$.
- *evalPercCov(AUT,$CSSC$)*: it evaluates the cumulative coverage statement percentage of AUT statements at the end of the process execution.

6.3.2 The Testing Infrastructure

The developed software infrastructure used to execute our testing process is now presented. This implementation targets the Android mobile platform.

The infrastructure includes two types of components, namely *Testing Process Coordinator* and *Testing Session Executor*. The former component is responsible for starting, ending, and managing the results of the testing sessions' execution. The latter component is in charge of running the random testing sessions on the emulated Android platforms and collecting data resulting from them. At run time, just a single instance of Testing Process Coordinator is needed, while many instances of the Testing Session Executor component can potentially be deployed and run on different nodes of a distributed architecture. Fig. 6.2 shows an example infrastructure, including two Testing Session Executor components.

Each Testing Session Executor component includes three software modules: Android Emulator, Driver and Loader. The Driver component implements the specific MFT technique and iteratively sends the next random user event to the GUI of the subject application. The Android Emulator provides the emulated execution platform and consists of the Android Virtual Device (AVD) provided by the Android SDK[3]. The instrumented Application Under Test is run on the AVD under the control of a Robot component that actually fires the event on the current GUI and scrapes it.

The AUT is instrumented by the EMMA Library[4] in order to generate Code Coverage Files. The Loader module fetches these files from the AVD and provides them to the Coverage Repository that is deployed on the Testing Process Coordinator component of the architecture. EMMA is an open-source toolkit for measuring and reporting Java code coverage. It supports coverage types such as class, method, line and basic block. Moreover, EMMA can detect when a single source code line is partially covered, which can happen when the source code has branches that are not exercised by the tests[5]. Referring to the algorithm in Fig. 2

[3]https://developer.android.com/sdk/index.html

[4]http://emma.sourceforge.net/index.html

[5]http://emma.sourceforge.net/faq.html#q.fractional.examples

the $i-th$ element of $SCS[]$ is actually the EMMA run-time coverage data (which basic blocks have been executed) that are stored in files having *.ec* extensions. To obtain the $CSSC$ the *merge* feature provided by EMMA was exploited.

As to the Process Coordinator component, it includes an Engine module that launches the testing sessions on the different Testing Session Executor components, gets their coverage results, and periodically assesses the termination condition. To evaluate the termination condition, it runs Emma scripts to compute the code coverage percentages reached both by the sessions and their union. It then executes scripts to evaluate whether the termination condition has been reached. While the termination condition is not true, the Engine commands the Driver to send further random events, otherwise it stops all running sessions.

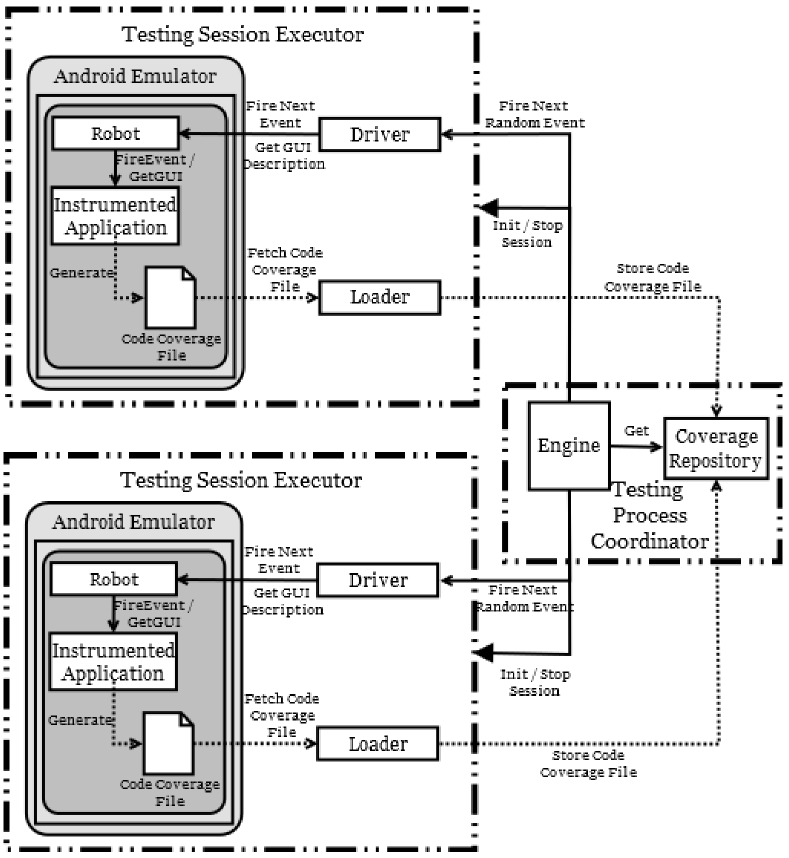


Figure 6.2: Overview of the overall testing infrastructure

This architecture was implemented using Java technologies as well as features provided by the Android Debug Bridge (ADB)[6].

Since the implementation of Driver and Robot modules depend on the MFT technique involved in the process, two versions of the Driver module were developed; they implement the fuzz testing technique used by the AndroidRipper tool [11] and the one exploited by the Android Monkey tool[7], respectively. The first version of the Driver directly delegates the Robot component to interact with the AUT by means of the APIs provided by the Android Instrumentation library[8]. Specifically, the Robot exploits the Robotium library[9] both to fire events on the AUT and to get an instance of its GUI at run-time. The second version of the Driver was implemented using the AndroidMonkey library[10]. It is a copy of the original Android Monkey Tool and is a library made for testing and analysis purposes. In this version, the Driver component runs test scripts that invoke the Robot, which in turn runs JUnit test cases and exploits the designated library to send event(s) to the AUT.

6.4 An exploratory study

This section presents a preliminary case study which assesses the feasibility of the proposed testing process and its capability of reaching the saturation effect. To this aim, the proposed testing process and infrastructure was used to test a real Android application named *SimplyDo*. This is a medium sized app (1,281 LOC) that provides a simple shopping and TODO list manager for Android. During the process, the MFT technique implemented by the AndroidRipper tool was ex-

[6]http://developer.android.com/tools/help/adb.html
[7]http://developer.android.com/tools/help/monkey.html
[8]http://developer.android.com/tools/testing/testing_android.html#Instrumentation
[9]http://code.google.com/p/robotium/
[10]https://code.google.com/p/androidmonkey/

ploited. In its basic configuration, this tool is able to fire events on GUI widgets having at least one event handler registered for the event with a delay of $1000ms$ between consecutive events.

In the first phase of the study, the process was performed multiple times. Each time the same AVD configuration but either different preconditions for the subject app or different MFT tool configurations were used. More specifically, two different tool configurations, C1 and C2, and two different app preconditions, P1 and P2 were defined. C1 and C2 are defined as follows:

- C1: Given a GUI with a ListView widget, Android Ripper fires 'click' events only on the first three items of the list.
- C2: Given a GUI with a ListView widget, Android Ripper fires 'click' events on all of the list items.

Similarly, the two preconditions, P1 and P2, are defined as:

- P1: SimplyDo has been installed on the device, but has never been launched.
- P2: SimplyDo has been launched, and contains two TODO lists and one item in the first TODOs list.

Table 6.1 reports the four combinations of tool settings and app preconditions considered for each process run.

Table 6.1: Testing process variants and results

TP	**MFTSet**	**AUTPr**	**TerL**	**TerP**
TP11	C1	P1	76.56%	4,000
TP12	C1	P2	76.32%	3,800
TP21	C2	P1	85.1%	4,700
TP22	C2	P2	84.63%	4,300

For each variant, $k = 12$ testing sessions in parallel were run, where each session was executed on a different PC. The automatically obtained $TerP$ and $TerL$ values for each variant are reported in Table 6.1. In order to obtain a more complete view of the trend of each process, up to 10,000 events for each session were ran to observe coverage even after the application had reached its termination point. Figure 6.3 illustrates the obtained statement coverage percentage trends for each of the variants.

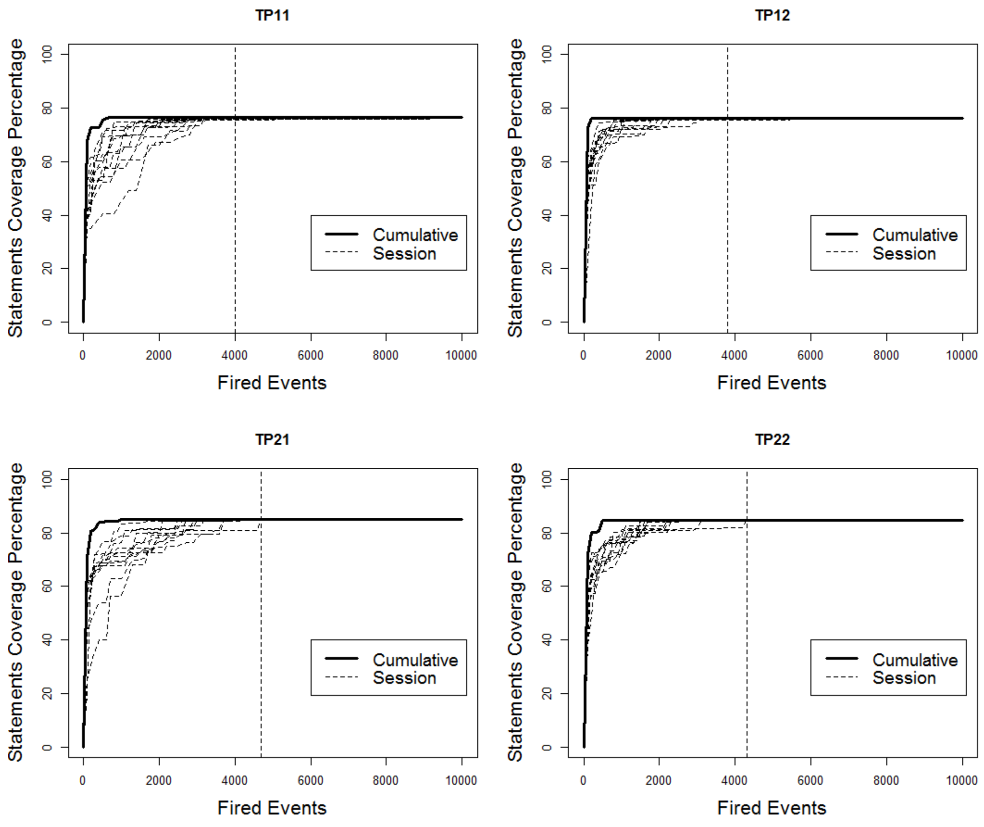


Figure 6.3: Coverage trends of the four testing process variants

The obtained trends seem to suggest that all processes reached the saturation effect. To confirm this datum, both the code coverage reports produced by the Java Code Coverage Tool EMMA and the part of the code of the application left out by the sessions was manually analyzed. The aim of the analysis was to decipher whether the resulting uncovered code was in fact reachable. If so, it would seem that the termination points were *not* all saturation points. On the other hand, if the uncovered code was found to be *unreachable*, could be concluded that every termination point of the process was also its saturation point and the hypotheses would still hold.

At the end of the analysis, can be observed that in fact all the uncovered statements could never be exercised by the testing processes due to the following reasons:

- **MFT tool configuration.** Some code was not reachable given the MFT tool configurations. For example, because AndroidRipper was not configured to fill in the EditText widgets with null values, the related *NullPointerException* handling code could not be covered. Moreover, since AndroidRipper did not use the keyboard device to fill the EditText widgets, the related *onEditorAction* handlers were actually unreachable. Eventually, the C1 settings used in TP11 and TP12 were able to fire events only on the first three items of any menu list, and then the code associated with the other menu items could never be triggered.

- **AVD configuration.** Some parts of the source code of the application could never be executed due to the settings of the AVD. For example, the code that should be executed when the device does not include the SD card was not reachable because the AVD used in the testing environment was equipped with an *emulated* SD card.

- **AUT preconditions.** Another set of statements were not reachable on the basis of the AUT preconditions. For example, in the process variants TP12 and TP22 that launched the app from the P2 precondition, three LOCs executing *CREATE TABLE* SQL queries were never exercised. These queries are executable only when the app is first launched after the installation.

- **Unreachable code of the AUT.** The remaining uncovered statements were unreachable, because there was no control flow path to it from the rest of the program [33]. As an example, can be found classes and methods included in the source code but never used by the rest of the program.

As a result, all reachable code of the AUT was covered by the testing processes, and therefore, saturation was *always* reached at a termination point of the process.

To further confirm the results, whether the choice of $k = 12$ sessions composing the process had influenced the obtained results was analyzed. To this aim, the coverage data and evaluated the termination points proposed by processes made of k=2, 3, 4 and 8 sessions were post-processed. The 12 testing sessions were permuted without repetitions obtaining 66 simulations of testing processes made by $k = 2$ testing sessions, 220 testing processes made by $k = 3$ testing sessions, 495 testing processes made by $k = 4$ testing sessions and 495 testing processes made by $k = 8$ testing sessions. The termination levels obtained were always the same for testing processes composed of $k > 2$ sessions, and conclude that $k = 12$ sessions is an adequate choice to obtain reliable process results.

Lastly, in order to assess whether the process is able to reach the saturation independently of the exploited MFT technique, another testing process was performed involving the same application but a different MFT technique implemented by Monkey Tool. It fires events belonging to different classes of User Events and System Events chosen at random on the basis of a given adjust percentage. User events includes touchscreen events that are fired on randomly chosen points of the screen irrespective of the actual presence of a UI widget in that point. A single testing process was executed made by twelve random testing sessions, starting from the initial P2 state of the AUT and configured the tool with its default adjust percentages. Moreover, the delay between consecutive events was set at $100ms$.

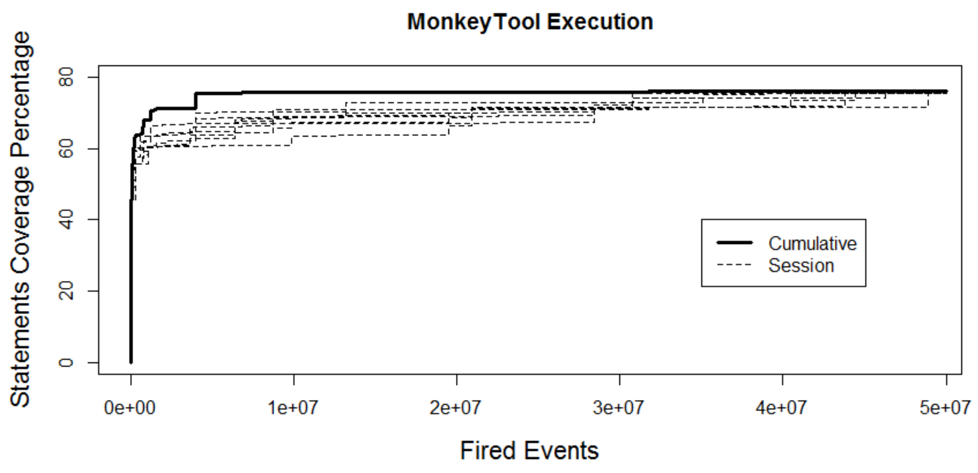


Figure 6.4: Coverage trends obtained by Monkey Tool

Figure 6.4 reports the Statement Coverage Percentage trend achieved by the process, where $TerP = 5 \times 10^7$ and $TerL = 75.57\%$. The manual analysis of the code statements left uncovered by the process confirms the uncovered code is in fact unreachable due to the same motivations listed above. This result, therefore, further confirms that the proposed process reached the saturation effect independently from the considered MFT technique, even if this effect was reached after many more events than the former process (after about 5×10^7 events rather than about $4,000$ events).

6.5 Experimentation

To extend the validity of the results achieved by the exploratory study, an experiment was conducted aimed at answering the two Research Questions reported below.

R.Q.1: Is the proposed testing process able to reach the Saturation at the termination point?

R.Q.2: Which are the main factors able to affect the effectiveness of the proposed testing process at the termination point?

These research questions were addressed in the context of Android mobile applications, using the MFT technique implemented by AndroidRipper.

6.5.1 Subjects

For this experiment 18 open source Android applications have been selected, published on the Google Play store, belonging to different Google Play categories and having different source code complexity, expressed in terms of number of statements. Table 6.2 reports for each application an identifier (**AUT ID**), its name, Google Play Category (GPC) and the number of source code statements given by $\mid SST \mid$. As data show the AUTs belonged to 12 different GPCs and their size varied from 184 to 3860 LOCs.

6.5.2 Metrics

To assess the Saturation effect at the termination point of the process the residual percentage of code statements left uncovered by the testing process until the termination point has been measured. If the Saturation is reached then this residual quantity is zero, or approximately equal to zero.

To evaluate this residual percentage, the following sets and metrics were used:

- $UnS(TP)$: it is the **Un**reachable **S**tatements set, $UnS(TP) \subseteq SST$ of the AUT, made of all the AUT statements that are unreachable by the process TP.

Table 6.2: Characteristics of the Applications Under Test

AUT ID	**AUT Name**	**GPC**	**—SST—**
AUT1	AardDict	Book	2,097
AUT2	AndroidLevel	Tools	623
AUT3	BatteryCircle	Tools	249
AUT4	BatteryDog	Tools	463
AUT5	Bites	Lifestyle	967
AUT6	Fillup	Transportation	3,807
AUT7	JustSit	Lifestyle	273
AUT8	ManPages	Productivity	292
AUT9	MunchLife	Entertainment	184
AUT10	NotificationPlus	Productivity	283
AUT11	Pedometer	Health	809
AUT12	QuickSettings	Productivity	2,841
AUT13	Taksman	Tools	226
AUT14	TicTacToe	Brain	493
AUT15	TippyTipper	Finance	999
AUT16	Tomdroid	Productivity	3,860
AUT17	Trolly	Shopping	364
AUT18	WorldClock	Travel	1,149

- $ReSRS(TP, TerP)$: it is the **Re**sidual **S**et of **R**eachable **S**tatements by TP at the termination point. It represents the set of statements that are potentially reachable by TP but have not been covered by it until $TerP$. This set is given by the following difference:

$$SST - CSSC(TP, TerP) - UnS(TP). \tag{6.5}$$

- $RePRS(TP, TerP)$: it is the **Re**sidual **P**ercentage of **R**eachable **S**tatements. It is given by equation (6.6)

$$RePRS(TP, TerP) = \frac{\mid ReSRS(TP, TerP) \mid}{\mid SST \mid} \times 100 \tag{6.6}$$

  If $RePRS(TP, TerP) = 0$ all the reachable code of the application that is actually covered by TP, so the process reached the saturation.

6.5.3 Experimental Procedure

The processes was configured to test each subject application. Each process included $k = 12$ sessions and the same AndroidRipper configuration to test all the AUTs. AndroidRipper was configured for sending events on the GUI widgets having at least a registered listener and for emulating the pressure both of the *back* button and of the *openMenu* one of the mobile device. The delay between two consecutive events was set to 500 ms and AndroidRipper was configured for filling in the EditText widgets with random numeric values. Moreover, for each AUT a specific initial precondition was defined and the same AVD configuration has been used in each process execution: the emulated devices were all equipped with Android Gingerbread (2.3.3) and have 512 MByte of RAM and 64 MByte of memory on emulated SD Card. The processes were executed by exploiting the testing infrastructure that was configured to run on 13 different PCs running Windows 7 64 Bit Operative System equipped with an Intel I5 3GHz processor and 4GB of RAM.

To answer the first research question, at the end of each process the $RePRS$ values was measured. To measure this metric the $UnS(TP)$ was evaluated set by means of a manual analysis of the statements uncovered by each TP. To answer the second research question, the effectiveness of the process at the termination point was assessed by evaluating its Termination Level $TerL$. Moreover, for the motivations that did not allow the complete coverage of the source code statements were analyzed. This research was made by manual analysis too.

To be confident about the results of the manual analysis, they were performed by two different teams of software engineers (each one including a Ph.D. student and a graduate student), and then the obtained results were validated by a third team including two researchers in software engineering.

6.5.4 Results

Table 6.3 reports for each AUT the termination level $TerL$ and the residual percentage of reachable statements $RePRS$ achieved by the performed testing processes.

6.5.4.1 Saturation results

As the data show, 15 times out of 18 the residual percentage of reachable statements was 0%. In the remaining three cases, this percentage was negligible, being lower then 0.5%. As to AUT3, the RePRS was 0.40%, indicating that a single line of code over 249 of its SST was potentially reachable, but it was not actually reached. As to AUT4, the residual percentage of reachable statements was 0.42% due to only two potentially executable but not actually executed statements over

Table 6.3: Experimental Results

AUT ID	**TerL**	**RePRS**	**AUT ID**	**TerL**	**RePRS**
AUT1	71.14%	0%	AUT10	41.24%	0%
AUT2	62.68%	0%	AUT11	74.75%	0%
AUT3	92.89%	0.40%	AUT12	48.50%	0%
AUT4	81.60%	0.42%	AUT13	92.79%	0%
AUT5	57.88%	0%	AUT14	99.64%	0.17%
AUT6	84.03%	0%	AUT15	87.86%	0%
AUT7	70.92%	0%	AUT16	69.96%	0%
AUT8	77.53%	0%	AUT17	80.88%	0%
AUT9	98.86%	0%	AUT18	97.37%	0%

463. In regards to AUT14, the residual percentage of reachable statements was even smaller, 0.17%, with only 0.8 potentially executable statements that were not covered during the process.

On the basis of these results **R.Q. 1** could be answered and can be concluded that the proposed testing process was able to reach the Saturation at the termination point in all the considered cases.

6.5.4.2 Effectiveness results

The uncovered code of the AUTs was analyzed in detail with the aim of understanding why this code was not reached during the process. The motivations reported below were found.

(1) *Part of the code was not reached depending on the initial state of the applications under test.* This motivation was true for two applications, namely Bites and FillUp. Since the considered preconditions set them in a state *successive to the first AUT execution on the device*, then the statements of some SQL queries needed to configure the supporting databases was actually unreachable. This code can be indeed executed only when these applications are launched for the first time

after their installation on the device.

(2) *Part of the code was not reached depending on the configuration of the device exploited in the testing processes.* This motivation was verified for 8 applications out of 18. Code statements were found that could be executed only if the apps were installed on specific devices, i.e., the ones belonging to the *Motorola* and *eInk Nook* families. Some other statements were unreachable because they can be executed only on Android O.S. platforms different from the 2.3.3 version. Other statements were made unreachable by the hardware limitations of the device emulator, i.e., the absence of sensors, WiMAX connectivity, Bluetooth and a physic LED and the impossibility of changing the connection status (Wi-Fi on/off, 3g on/off). Part of the source code was unreachable because some specific apps, like GMail, were not present on the device emulator. Eventually, some statements could not be reached because they can be triggered only when the device does not present an SD card, while the emulator was equipped with an SD.

(3) *Part of the code was not reached depending on both the settings and the limitations of the MFT tool.* This motivation was verified for 11 applications out of 18. The MFT tool was configured to fill in the editText widgets with random integer values. As a consequence, some statements whose execution requires specific user input values (such as, a valid e-mail address or valid URLs) could never be covered. Parts of the statements were actually unreachable, given the limitations of the MFT tool that is not able to fire all the types of event handled by the subject apps. As an example, AndroidRipper is not able to emulate the pressure of some device buttons (such as Volume Up, Volume Down and Search), to interact with the device Trackball, to fire specific events like the gesture ones, to emulate the changes of the values read by sensors, to send Intents, to interact with some widgets like Preferences and WebViews.

(4) *Part of the code was not reached because it is actually unreachable code of*

the AUT. This motivation was verified for all the considered applications. Some apps presented activities declared in the AndroidManifest.xml but never opened, menus defined but never enabled, or classes that are never instantiated. In an application, there was a fraction of source code related to the creation of XML files that could never be executed since the AUT lacked of the permits for writing on the SD Card. In some applications, there were parts of code related to the interaction with external services that are not actually available.

In conclusion the motivations that emerged from this analysis coincided with the same ones revealed by the exploratory study, so **R.Q. 2** could be answered by claiming that the main factors affecting the effectiveness of the proposed testing processes were (1) *the preconditions of the AUT*, (2) *the configurations of the testing platform*, (3) *the limitations and the configuration of the MFT technique*, and (4) *the existence of unreachable statements in the source code of the AUT.*

6.5.5 Lessons learned

At the end of the experiment, some lessons about the proposed testing process can be learned. A first lesson regards its termination criterion. According to it, in the experiment the termination points were reached when each session composing the process reached the same code coverage as the cumulative one. This stop condition allowed the process to reach the saturation effect. Analyzing the experimental data further, an alternative stop condition can be derived. The data suggested indeed that the process could be stopped, without loss of code coverage, as soon as the coverage of one of the sessions reached the cumulative one. As an example, Fig. (6.5) shows a zoom in the code coverage trends achieved by the testing process TP22 presented in Section 6.4 and highlights two possible termination points $TerP$ and $TerP^*$, where the second point is obtained by using the new termination condition $TerCond^*$ expressed by equation (6.7):

$$\begin{aligned} &TerCond^*(S,n) = true \iff \\ &\exists\, S_i \in S \mid CSP(S_i,n) == CCSP(TP,n) \end{aligned} \tag{6.7}$$

To validate this intuition, the new Termination points and levels that could be achieved by the new termination condition for each AUT were evaluated. Table 6.4 shows the obtained results. The termination levels obtained using the new termination condition were the same as the ones reported in Table 6.3. As the data show, the new termination condition significantly reduces the number of events needed to reach the same test adequacy as the one achieved at the saturation point, and the reduction rate varies between 24.48% and 93.58%. This new criterion may be successfully used to improve the efficiency of the process, leaving its test adequacy unchanged.

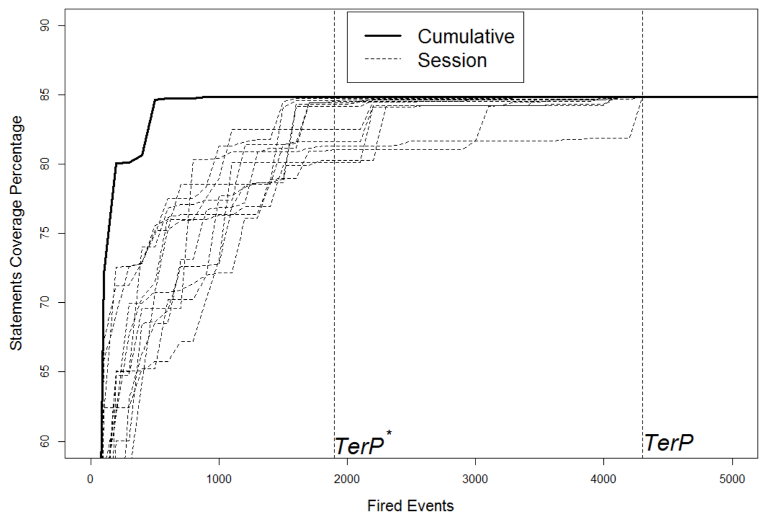


Figure 6.5: Two termination points of TP22 execution

A second lesson can be learned from the exploratory study is that the proposed process is able to reach different Saturation Levels, depending on the preconditions of the AUT and the settings of the MFT technique. This lesson suggests to us a

Table 6.4: Termination Points

AUT ID	$TerP^*$	$TerP$	**AUT ID**	$TerP^*$	$TerP$
AUT1	300	4000	AUT10	100	500
AUT2	100	700	AUT11	2300	4600
AUT3	100	500	AUT12	15200	30700
AUT4	100	800	AUT13	300	800
AUT5	7400	9800	AUT14	10300	20100
AUT6	53400	95600	AUT15	500	7200
AUT7	1900	5300	AUT16	98200	147800
AUT8	500	7800	AUT17	200	600
AUT9	300	700	AUT18	1400	3600

new variant of the testing process that iteratively selects different app-tool configurations until the saturation effect is detected. As an example, the tester may create a set of app- and tool- configurations. The new process iteratively picks a configuration and runs the MFT tool until the saturation point. The process ends when the set of app- and tool- configurations is exhausted. This new process may achieve better test adequacy results than the former one. Fig. (6.6) shows the cumulative coverage trend obtained by the new testing process where were consecutively ran the four testing variants TP12, TP11, TP22 and TP21 described in Section 6.4. However, further experiments should be performed in order to assess the validity of these two process variants.

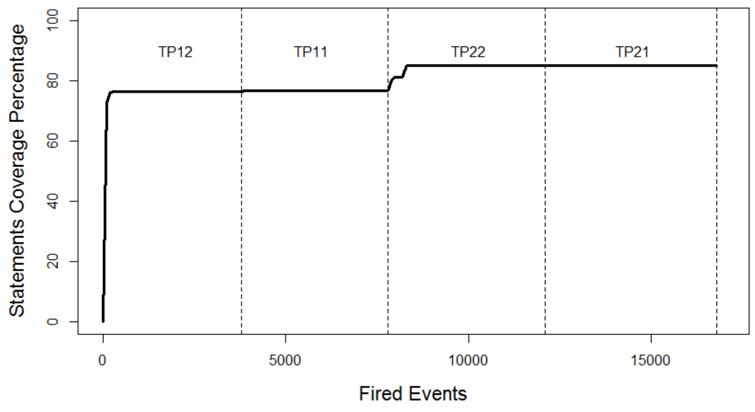


Figure 6.6: Cumulative Coverage achieved in the new process variant

CHAPTER 7

A parallel and distributed implementation of GUI Ripping Techniques

In this chapter I present a parallel approach for automated GUI Testing of Android Applications; the approach is based on the generic algorithm described in Chapter 3 and is implemented exploiting the *Android Ripper* tool.

7.1 Introduction

Testing applications that can assume a large set of possible GUI states may need a large number of test cases and a large amount of testing time. Contributions in the literature about automated GUI Testing of Android applications rarely address the problem of the efficiency of the process; their primary focus is often the effectiveness of the process or the mitigation of the manual effort. In Chapter 3 the experimentation carried out on *Android Ripper* needed many hours to be completed. This time can grow due to the size and complexity of the GUI and the number of user events to trigger [90].

To address the problem of the efficiency of the testing process presented in Chapter 3, in this Chapter a parallel and distributed approach is presented. In literature there are some contributions that propose a process to execute test cases simultaneously. Hu et al. [52] propose a technique focused on the parallel execution of the Monkey tool in a distributed environment. Nguyen et al. [90] present GUITAR, a tool implementing an automated testing process that can distribute test cases to a cluster to slave nodes. Amalfitano et al. [5] propose a process to determine an optimal termination point for Random techniques that exploits the parallel execution of twelve testing sessions. Wen et al. [114], instead, propose PATS (Parallel Android Testing System), a parallel GUI testing platform which performs GUI testing based on a master-slave model; not only it executes GUI events on the AUT, but it analyzes the GUI dynamically under the cooperation of the master and the slaves.

7.2 A parallel implementation of Android Ripper

Algorithm 1 described in Chapter 3 is a generic algorithm; it has been implemented in the *Android Ripper* tool that is based on a Master-Slave model. To distribute the process among many Slave nodes different needs should be addressed both for Active Learning and Random techniques.

As regards Random techniques, a Slave node executes sequences of pseudo-random events where a new event is scheduled starting from the status of the GUI of the AUT reached by performing all the previous sequence of events. In this scenario, Random techniques can be parallelized by executing simultaneously different instances of the *RandomDriver* (see Subsection 3.2.3), one for each Slave node. Our implementation of such solution is a multi-threaded version of the *RandomDriver*.

For Active Learning techniques, instead, a Slave node executes a predefined sequence of events and may update the *AppModel* at the end of its execution. In this case, a sequence of events should be assigned at one Slave node only and

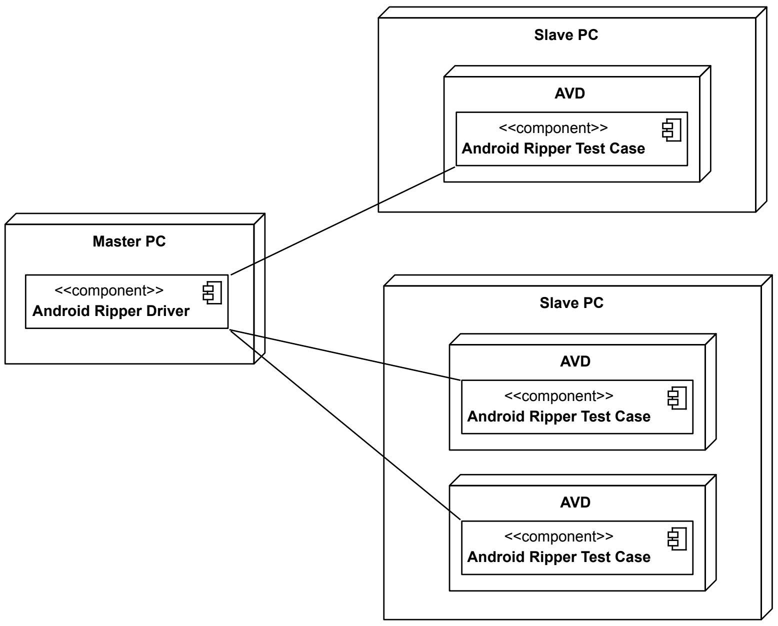


Figure 7.1: AndroidRipper Master/Multi-Slave Implementation

the *AppModel* should always be updated coherently. To this matter the implementation of *RefineAppModel* and *RunEvents* methods have been modified. The *RunEvents* method has been implemented to control different Slave nodes at a time and, after the execution of a sequence of events, the *RefineAppModel* is run. This method modifies the *AppModel* and this modification should be done in mutual exclusion. So, a semaphore has been implemented to control the access to this shared resource.

Figure 7.1 show an high-level architecture of the system.

Slave nodes are meant to be distributed among different PCs and one PC can run one or more Slave nodes at a time. The communication between the *Driver* and a *Slave* node is implemented by exploiting a JSON/RPC-based protocol over a TCP/IP connection.

Figure 7.2 exemplifies the interaction between a *Driver* component, that act as a Master node, and two **AndroidRipper Test Case** components, that act as

Slave nodes. The names of the methods are coherent with the ones used for the generic algorithm.

The figure shows how, after that the sequences of events are scheduled (*scheduleEvents()*), two different sequences of events are executed in parallel (*fireEvents()*); the Figure evidences that the *refine()* method of the *AppModel* is executed in a critical region.

7.3 Case Study

In this section a case study is described where a comparison is made between the original Master/Slave and the proposed Master/Multi-Slave approaches. The experimental results are obtained by the execution of an Active Learning technique and are analyzed in detail. The exploited Active Learning technique features the following configuration:

- **TerminationCriterion:** ModelCoverageTerminationCriterion
- **ExtractionCriterion:** RelevantEventsExtractionCriterion
- **AbstractionStrategy:** TAVAbstractionStrategy
- **SchedulingStrategy:** BreadthSchedulingStrategy

7.3.1 Research Questions

The case study is aimed at comparing the effectiveness and the efficiency of the proposed Master/Multi-Slave approach with respect to the original Master/Slave one. In detail, the case study want to address the following research questions:

- **RQ1:** Is the effectiveness of the considered technique influenced by the parallel implementation?

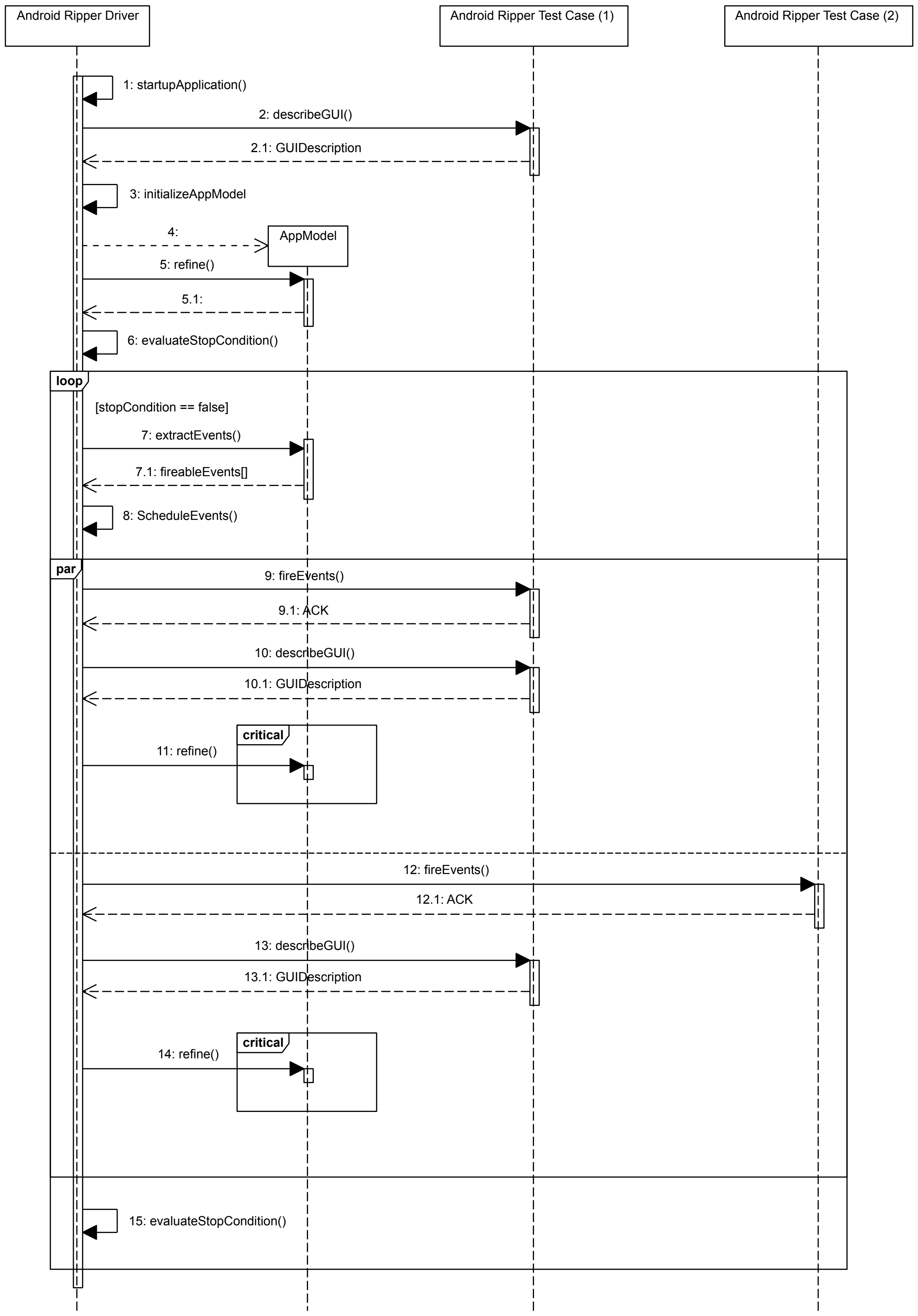


Figure 7.2: AndroidRipper Master/Multi-Slave Implementation

- **RQ2:** How the quantity and the distribution of Slave nodes on different machines influence the effectiveness of the parallel implementation of the considered technique?
- **RQ3:** Is the efficiency of the Active Learning technique influenced by the parallel implementation?
- **RQ4:** How the quantity and the distribution of Slave nodes on different machines influence the efficiency of the parallel implementation of the considered technique?

7.3.2 Variables & Measures

In this Subsection the variables of the experiment are described: the independent variables, i.e. the variables that are changed or manipulated during the experiment, and the dependent variables, i.e. the values that depend from the variation of the independent variables.

7.3.2.1 Independent Variables

Given that more than one Slave node can be deployed on different virtual devices running on a single Machine, n_S is defined as the number of Slave nodes and n_M as the number of Machines where the Slave nodes are allocated. The independent variable of the experiment is the *Configuration* that is a couple (n_M,n_S). In the experiment the following values for this variable are used: {(1,1); (2,1); (1,2); (2,2); (6,1); (6,2)}, where the configuration (1,1) corresponds to the non-parallel Master/Slave implementation of the Active Learning technique.

7.3.2.2 Dependent Variables

The dependent variables of the experiment are related to effectiveness and efficiency. To measure the effectiveness for the testing techniques the **LOCs Coverage Percentage (COV%)** is evaluated; COV% is the percentage of lines of the source code of the application covered during the testing process. To measure the efficiency of the testing technique the **Testing Process Execution Time** (t_{TOT}) is measured; t_{TOT} is expressed in hours and minutes (hh:mm) and includes the Start-up time of each Slave node and the time to execute each sequence of events generated during the process. Configurations with more than two Slave nodes on the same Machine are not considered due to resource limits of the Machines.

7.3.3 Experiment Setup

7.3.3.1 Objects of the Experiment

As objects of the experiment a sample of 3 open source Android apps were selected from the Google Play market. Table 7.1 describes the selected applications and for each app reports its Identifier, a short Description, and software structural metrics related to its source code.

Table 7.1: Applications Characteristics

ID	Application	Description	# Classes	# Activ.	# Methods	# Event handlers	# LOCs
AUT1	TicTacToe 1.0	Simple game	13	1	47	16	493
AUT2	TippyTipper 1.2	Tip calculator app	42	6	225	70	999
AUT3	Tomdroid 0.7.1	Note-taking app	133	10	707	117	3860

The metrics include: total number of classes, number of Activity classes (i.e. classes extending the Activity class provided by the Android framework and that are responsible for implementing the GUIs of the application), number of methods, number of event handlers (i.e. methods responsible for managing the events sent to the app) and the number of lines of code (LOCs) of the app.

7.3.3.2 Experimental Procedure

In order to execute the experiment, the following three steps were performed.

- **Set-up of a testing environment.** The parallel testing techniques was implemented in Android Ripper. The *Driver* component was deployed on a single PC and we configured 6 PCs to run Slave nodes; both the Master and the Slave PCs are on the same network infrastructure, featuring a fixed network configuration. On each Slave PC two Android Virtual Devices (AVDs) were prepared to emulate devices having 512 MByte of RAM, a 64 MByte SD Card, and an Android Gingerbread (2.3.3) operating system.

- **Execution of the testing techniques.** Using the testing infrastructure all the 3 applications were tested. Since the application preconditions could affect the results of executing a testing technique, each app was set in the same pre-conditions before running each testing session. The testing techniques ran until they reached their termination point.

- **Data Collection.** After the termination of each testing process all the needed measures were performed by exploiting the raw data stored on the *Driver* PC. As regards the **LOCs Coverage Percentage (COV%)** the code coverage reports produced by Emma were collected. The **Testing Process Execution Time** (t_{TOT}) was directly measured by the *Driver*.

7.3.4 Results & Discussion

Table 7.2 reports the results obtained for the experiment.

Table 7.2: Experiment Results

		(n_M,n_S)					
		(1,1)	**(1,2)**	**(2,1)**	**(2,2)**	**(6,1)**	**(6,2)**
AUT1	**COV%**	19	19	19	19	19	19
	t_{TOT}	00:19	00:13	00:11	00:08	00:07	00:04
AUT2	**COV%**	58	58	58	58	58	58
	t_{TOT}	01:02	00:45	00:43	00:32	00:19	00:14
AUT3	**COV%**	32	32	32	32	32	32
	t_{TOT}	02:29	01:45	01:11	00:49	00:33	00:20

Looking at the collected data *RQ1* can be answered as follows: *the effectiveness of the considered technique* ***is not*** *influenced by the parallel implementation.* So, to answer to *RQ2*: *the quantity and the distribution of Slave nodes on different machines* ***do not*** *influence the effectiveness of the parallel implementation of the considered technique.*

As regards *RQ3*: *the efficiency of the Active Learning technique* ***is*** *influenced by the parallel implementation.* This implies that *the quantity and the distribution of Slave nodes on different machines* ***do*** *influence the effectiveness of the parallel implementation of the considered technique.* To answer *RQ4* two questions need to be answered:

- **RQ4.1:** How the number of Slave nodes (n_S) running on a single Machine influences the efficiency of the parallel GUI testing process?
- **RQ4.2:** How the number of Machines (n_M) running the same number of Slave nodes influences the efficiency of the parallel GUI testing process?

Table 7.3 shows an index of the speed-up of the technique related to the number of Slave nodes (n_S) running on each Machine; this index is evaluated as the inverse

Table 7.3: Performance varying n_S

	(n_M,n_S)		
	$\frac{(1,2)}{(1,1)}$	$\frac{(2,2)}{(2,1)}$	$\frac{(6,2)}{(6,1)}$
AUT1	1.46	1.38	1.75
AUT2	1.38	1.34	1.36
AUT3	1.42	1.45	1.65
AVERAGE	1.42	1.39	1.59
	AVERAGE = 1.46		

of the ratio of the t_{TOT} measured for configurations featuring the same n_M but different n_S.

It was expected that the speed-up grows at least linearly with respect to the number of Slave nodes running on the same Machine; instead in average the speed-up is only of 1.46 in correspondence of a doubling of the number of Slave nodes. So, referring to *RQ4.1*, can be stated that *the t_{TOT} does not linearly decrease with respect to the number of Slave nodes (n_S) running on each Machine.*

Table 7.4 shows an index of the speed-up of the technique related to the number of Machines (n_M) running the same number of Slave nodes; this index is evaluated as the inverse of the ratio of the t_{TOT} measured for configurations featuring the same n_S but different n_M.

It was expected that the speed-up grows at least linearly with respect to the number of Machines running the same number of Slave nodes; on the contrary, when using two Machines in average the speed-up is only of 1.74 instead of the optimal value of 2; using six Machine the increase was in average *3.70*. So, answering to *RQ4.2*, can be stated that *the t_{TOT} does not linearly decrease with respect to the number of Machines (n_M) running the same number of Slave nodes.*

From this case study some intuitions can be elicited. In general, when the

Table 7.4: Performance varying n_M

	(n_M,n_S)			
	$\frac{(2,1)}{(1,1)}$	$\frac{(2,2)}{(1,2)}$	$\frac{(6,1)}{(1,1)}$	$\frac{(6,2)}{(1,2)}$
AUT1	1.73	1.63	2.71	3.25
AUT2	1.44	1.41	3.26	3.21
AUT3	2.10	2.14	4.52	5.25
AVERAGE	1.76	1.72	3.50	3.90
	AVERAGE = 1.74		**AVERAGE = 3.70**	

number of Slave nodes is increased, independently from the number of Machines involved, the presence of delays caused both by the communication between the Slave nodes and the Driver and by the synchronization of the Slave nodes guided by the Driver, may reduce the optimal efficiency of the parallel technique. In particular, when deploying more than one Slave node on the same Machine the time needed to execute the testing process is probably influenced also by the presence of shared resources between the nodes like memory, CPU, disk, files and so on.

When comparing the results obtained by executing the process on more than one Machine, a speed-up of 174% was obtained when using two Machines; using six machines, instead, produced a speed-up of 370%, that is farther from the optimal value with respect of the result obtained by using two Machines. This may mean that when the number of Machines is increased the delay due to the communication and the synchronization weights more and more on the efficiency of the process. However, the parallelism has sped-up the testing process.

CHAPTER 8

Conclusions & Future Work

In this thesis a set of completely automated techniques supporting the testing of Android applications is presented, with the aim of reducing the manual effort needed by human testers. Firstly, the details of the design and the implementation of a novel automated GUI testing tool, called *Android Ripper*[1], are presented. Then four contributions related to the improvement of the performance of the techniques implemented in the tool are described. In detail the techniques presented are:

- a technique able to exercise context-sensitive applications [120];
- a search based testing technique applicable to Android applications with the purpose to generate test suites that are both effective in terms of coverage of the source code and efficient in terms of number of generated test cases;
- a technique to address the problem of stopping a random testing process at a cost-effective point;
- a parallel approach for automated GUI Testing of Android Applications.

Using the Android Ripper tool some experiments and case studies have been carried out to assess the validity of each approach. The results obtained witness

[1]https://github.com/reverse-unina/AndroidRipper

the effectiveness and the efficiency of the presented automated testing techniques.

To further assess the usefulness and the benefits of the contributions of this thesis, in the future, a case study experiment can be imagined involving testers performing real testing activities. Also, some future contributions could regard the proposal of hybrid techniques that include features both of Random and Active Learning techniques with the aim of obtaining technique that are both effective and efficient.

As regards the ideas presented in Chapter 4 where an extended version of Android Ripper able to effectively test context-sensitive applications is presented, some future works can be addressed. In the future an event-patterns repository can be build by analyzing a large corpus of bug reports related to mobile apps and to implement tools supporting the automatic injection of event patterns in existing test cases. As regards technological aspects of Android applications, events causing the interaction between different components of the same application or different applications can be considered, by adding to the Extended Ripper new features supporting Intent Messages generation and execution. Finally, wider experimentation can be addressed in order to assess the effectiveness of the proposed techniques also in terms of fault-detection.

One of the objectives of the future experimentation involving the AGRippin approach, presented in Chapter 5 will be the tuning of the algorithm parameters values (such as the crossover ratio, the mutation ratio, the number of iterations and the test suite size) and the evaluation of their influence on the effectiveness of the generated test suites and on the number of iterations needed to reach this level of effectiveness. As regards the crossover and mutation ratio, a technique can be implemented based on adaptive variations (as the one proposed by Srinivas and Ptnaik in [105]) in order to reduce the probability that the generated test suites maintain the same coverage for many consecutive iterations, as experienced in some of the case studies. As regards the number of iterations, longer experiments can be carried out in order to evaluate if some phenomena of convergence of the coverage to a global maximum may be observed. Finally, a variant of the technique can be tested that increases the test suite size when new test cases are generated by the Hill Climbing technique in order to avoid the loss of important test cases due to the selection operator.

In the future, the experimentation of the approach presented in Chapter 6, that addresses the problem of finding an optimal termination point for MFT tools, can be improved in order to extend the validity of the proposed technique. In particular, the approach can be exploited for testing a wider number of AUTs by means of other MFT tools even for different mobile operating systems and in other contexts, such as web or desktop applications.

As regards the parallel implementation of Android Ripper presented in Chapter 7, it can be modified to be deployed on a Cloud environment. Using a Cloud environment with a huge number of nodes, a wider experimentation can also be performed.

Bibliography

[1] G. D. Abowd, A. K. Dey, P. J. Brown, N. Davies, M. Smith, and P. Steggles. Towards a better understanding of context and context-awareness. In *Proceedings of the 1st International Symposium on Handheld and Ubiquitous Computing*, HUC '99, pages 304–307, London, UK, UK, 1999. Springer-Verlag.

[2] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, New York, NY, USA, 1st edition, 2010.

[3] W. Afzal, R. Torkar, and R. Feldt. A systematic review of search-based testing for non-functional system properties. *Inf. Softw. Technol.*, 51(6):957–976, June 2009.

[4] S. Ali, L. Briand, H. Hemmati, and R. Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *Software Engineering, IEEE Transactions on*, 36(6):742–762, Nov 2010.

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

[6] D. Amalfitano, N. Amatucci, A. R. Fasolino, U. Gentile, G. Mele, R. Nardone, V. Vittorini, and S. Marrone. Improving code coverage in android apps testing by exploiting patterns and automatic test case generation. In *Proceedings of the 2014 International Workshop on Long-term Industrial Collaboration on Software Engineering*, WISE '14, pages 29–34, New York, NY, USA, 2014. ACM.

[7] D. Amalfitano, N. Amatucci, A. R. Fasolino, and P. Tramontana. A conceptual framework for the comparison of fully automated gui testing techniques. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, pages 50–57, Nov 2015.

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

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

[10] D. Amalfitano, A. Fasolino, P. Tramontana, S. De Carmine, and G. Imparato. A toolset for gui testing of android applications. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 650–653, Sept 2012.

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

[12] D. Amalfitano, A. R. Fasolino, S. D. Carmine, A. Memon, and P. Tramontana. Using gui ripping for automated testing of android applications. In *ASE '12: Proceedings of the 27th IEEE international conference on Automated software engineering*, Washington, DC, USA, 2012. IEEE Computer Society.

[13] D. Amalfitano, A. R. Fasolino, P. Tramontana, and B. Robbins. Testing android mobile applications: Challenges, strategies, and approaches. *Advances in Computers*, 89:1–52, 2013.

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

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

[16] A. Arcuri, M. Iqbal, and L. Briand. Random testing: Theoretical results and practical implications. *Software Engineering, IEEE Transactions on*, 38(2):258–277, March 2012.

[17] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. *SIGPLAN Not.*, 48(10):641–660, Oct. 2013.

[18] G. Bae, G. Rothermel, and D. H. Bae. On the relative strengths of model-based and dynamic event extraction-based gui testing techniques: An empirical study. In *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*, pages 181–190, Nov 2012.

[19] J. E. Baker. Adaptive selection methods for genetic algorithms. In *Proceedings of the 1st International Conference on Genetic Algorithms*, pages 101–111, Hillsdale, NJ, USA, 1985. L. Erlbaum Associates Inc.

[20] M. Baldauf, S. Dustdar, and F. Rosenberg. A survey on context&#45;aware systems. *Int. J. Ad Hoc Ubiquitous Comput.*, 2(4):263–277, June 2007.

[21] I. Banerjee, B. Nguyen, V. Garousi, and A. Memon. Graphical user interface (gui) testing: Systematic mapping and repository. *Inf. Softw. Technol.*, 55(10):1679–1694, Oct. 2013.

[22] A. Barbosa, A. C. Paiva, and J. C. Campos. Test case generation from mutated task models. In *Proceedings of the 3rd ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '11, pages 175–184, New York, NY, USA, 2011. ACM.

[23] F. Belli, M. Beyazit, and A. Memon. Testing is an event-centric activity. In *Proceedings of the 2012 IEEE Sixth International Conference on Software Security and Reliability Companion*, SERE-C '12, pages 198–206, Washington, DC, USA, 2012. IEEE Computer Society.

[24] T.-H. Chang, T. Yeh, and R. C. Miller. Gui testing using computer vision. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 1535–1544, New York, NY, USA, 2010. ACM.

[25] T. Chen, F. Kuo, R. Merkel, and T. Tse. Adaptive random testing: The art of test case diversity. *JSS*, 83(1):60–66, 2010.

[26] T. Y. Chen, H. Leung, and I. K. Mak. Adaptive random testing. In *Proceedings of the 9th Asian Computing Science Conference on Advances in Computer Science: Dedicated to Jean-Louis Lassez on the Occasion of His*

5th Cycle Birthday, ASIAN'04, pages 320–329, Berlin, Heidelberg, 2004. Springer-Verlag.

[27] W. Choi, G. Necula, and K. Sen. Guided gui testing of android apps with minimal restart and approximate learning. *SIGPLAN Not.*, 48(10):623–640, Oct. 2013.

[28] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. Experimental assessment of random testing for object-oriented software. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 84–94, New York, NY, USA, 2007. ACM.

[29] I. Ciupa, A. Pretschner, A. Leitner, M. Oriol, and B. Meyer. On the predictability of random tests for object-oriented software. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 72–81, April 2008.

[30] I. Ciupa, A. Pretschner, M. Oriol, A. Leitner, and B. Meyer. On the number and nature of faults found by random testing. *Softw. Test., Verif. Reliab.*, 21(1):3–28, 2011.

[31] J. Clarke, J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd. Reformulating software engineering as a search problem. *Software, IEE Proceedings -*, 150(3):161–175, June 2003.

[32] L. Corral and I. Fronza. Better code for better apps: A study on source code quality and market success of android applications. In *Mobile Software Engineering and Systems (MOBILESoft), 2015 2nd ACM International Conference on*, pages 22–32, May 2015.

[33] S. K. Debray, W. Evans, R. Muth, and B. De Sutter. Compiler techniques for code compaction. *ACM Trans. Program. Lang. Syst.*, 22(2):378–415, Mar. 2000.

[34] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *IEEE Trans. Softw. Eng.*, 10(4):438–444, July 1984.

[35] S. Elbaum, G. Rothermel, S. Karre, and M. Fisher II. Leveraging user-session data to support web application testing. *IEEE Trans. Softw. Eng.*, 31(3):187–202, Mar. 2005.

[36] W. Enck, M. Ongtang, and P. McDaniel. Understanding android security. *Security Privacy, IEEE*, 7(1):50–57, Jan 2009.

[37] J. E. Forrester and B. P. Miller. An empirical study of the robustness of windows nt applications using random testing. In *Proceedings of the 4th Conference on USENIX Windows Systems Symposium - Volume 4*, WSS'00, pages 6–6, Berkeley, CA, USA, 2000. USENIX Association.

[38] C. A. Furia, B. Meyer, M. Oriol, A. Tikhomirov, and Y. Wei. The search for the laws of automatic random testing. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1211–1216, New York, NY, USA, 2013. ACM.

[39] S. R. Garzon and D. Hritsevskyy. Model-based generation of scenario-specific event sequences for the simulation of recurrent user behavior within context-aware applications (wip). In *Proceedings of the 2012 Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium*, TMS/DEVS '12, pages 29:1–29:6, San Diego, CA, USA, 2012. Society for Computer Simulation International.

[40] U. Gentile, S. Marrone, G. Mele, R. Nardone, and A. Peron. Test specification patterns for automatic generation of test sequences. In F. Lang and F. Flammini, editors, *Formal Methods for Industrial Critical Systems*, volume 8718 of *Lecture Notes in Computer Science*, pages 170–184. Springer International Publishing, 2014.

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

[42] T. Griebe and V. Gruhn. A model-based approach to test automation for context-aware mobile applications. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 420–427, New York, NY, USA, 2014. ACM.

[43] D. Hackner and A. M. Memon. Test case generator for GUITAR. In *ICSE '08: Research Demonstration Track: International Conference on Software Engineering*, Washington, DC, USA, 2008. IEEE Computer Society.

[44] D. Hamlet. When only random testing will do. In *Proceedings of the 1st International Workshop on Random Testing*, RT '06, pages 1–9, New York, NY, USA, 2006. ACM.

[45] R. Hamlet. Random testing. In *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

[46] R. Hamlet. *Random Testing*. John Wiley & Sons, Inc., 2002.

[47] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan. Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps.

In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, pages 204–217, New York, NY, USA, 2014. ACM.

[48] M. Harman, U. Ph, and B. F. Jones. Search-based software engineering. *Information and Software Technology*, 43:833–839, 2001.

[49] J. H. Hicinbothom and W. W. Zachary. A tool for automatically generating transcripts of human-computer interaction. *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, 37(15):1042, 1993.

[50] G. Holzmann. *Spin Model Checker, the: Primer and Reference Manual*. Addison-Wesley Professional, first edition, 2003.

[51] C. Hu and I. Neamtiu. Automating gui testing for android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST '11, pages 77–83, New York, NY, USA, 2011. ACM.

[52] C. Hu and I. Neamtiu. Automating gui testing for android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST '11, pages 77–83, New York, NY, USA, 2011. ACM.

[53] G. Hu, X. Yuan, Y. Tang, and J. Yang. Efficiently, effectively detecting mobile app bugs with appdoctor. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 18:1–18:15, New York, NY, USA, 2014. ACM.

[54] IDC. Smartphone os market share, 2015 q2, 2015.

[55] C. S. Jensen, M. R. Prasad, and A. Møller. Automated testing with targeted event sequence generation. In *Proceedings of the 2013 International Sym-*

posium on Software Testing and Analysis, ISSTA 2013, pages 67–77, New York, NY, USA, 2013. ACM.

[56] M. Joorabchi, A. Mesbah, and P. Kruchten. Real challenges in mobile app development. In *Empirical Software Engineering and Measurement, 2013 ACM / IEEE International Symposium on*, pages 15–24, Oct 2013.

[57] J. Kaasila, D. Ferreira, V. Kostakos, and T. Ojala. Testdroid: Automated remote ui testing on android. In *Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia*, MUM '12, pages 28:1–28:4, New York, NY, USA, 2012. ACM.

[58] P. Kapur, H. Pham, A. Gupta, and P. Jha. Testing-coverage and testing-domain models. In *Software Reliability Assessment with OR Applications*, Springer Series in Reliability Engineering, pages 131–170. Springer London, 2011.

[59] H. Kim, B. Choi, and W. Wong. Performance testing of mobile applications at the unit test level. In *Secure Software Integration and Reliability Improvement, 2009. SSIRI 2009. Third IEEE International Conference on*, pages 171–180, July 2009.

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

[61] A. Kumar Maji, K. Hao, S. Sultana, and S. Bagchi. Characterizing failures in mobile oses: A case study with android and symbian. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, pages 249–258, Nov 2010.

[62] X. Li, Y. Jiang, Y. Liu, C. Xu, X. Ma, and J. Lu. User guided automation for testing mobile apps. In *Software Engineering Conference (APSEC), 2014 21st Asia-Pacific*, volume 1, pages 27–34, Dec 2014.

[63] Y.-D. Lin, E. T. H. Chu, S.-C. Yu, and Y.-C. Lai. Improving the accuracy of automated gui testing for embedded systems. *IEEE Software*, 31(1):39–45, Jan 2014.

[64] Y. D. Lin, J. F. Rojas, E. T. H. Chu, and Y. C. Lai. On the accuracy, efficiency, and reusability of automated test oracles for android devices. *IEEE Transactions on Software Engineering*, 40(10):957–970, Oct 2014.

[65] C.-H. Liu, C.-Y. Lu, S.-J. Cheng, K.-Y. Chang, Y.-C. Hsiao, and W.-M. Chu. Capture-replay testing for android applications. In *Computer, Consumer and Control (IS3C), 2014 International Symposium on*, pages 1129–1132, June 2014.

[66] Z. Liu, X. Gao, and X. Long. Adaptive random testing of mobile application. In *Computer Engineering and Technology (ICCET), 2010 2nd International Conference on*, volume 2, pages V2–297–V2–301, April 2010.

[67] M. R. Lyu, editor. *Handbook of Software Reliability Engineering.* McGraw-Hill, Inc., Hightstown, NJ, USA, 1996.

[68] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 224–234, New York, NY, USA, 2013. ACM.

[69] R. Mahmood, N. Mirzaei, and S. Malek. Evodroid: Segmented evolutionary testing of android apps. In *Proceedings of the 22Nd ACM SIGSOFT In-*

ternational Symposium on Foundations of Software Engineering, FSE 2014, pages 599–609, New York, NY, USA, 2014. ACM.

[70] P. Maiya, A. Kanade, and R. Majumdar. Race detection for android applications. *SIGPLAN Not.*, 49(6):316–325, June 2014.

[71] Q. Malik, J. Lilius, and L. Laibinis. Scenario-based test case generation using event-b models. In *Advances in System Testing and Validation Lifecycle, 2009. VALID '09. First International Conference on*, pages 31–37, Sept 2009.

[72] S. Marrone, F. Flammini, N. Mazzocca, R. Nardone, and V. Vittorini. Towards model-driven v&v assessment of railway control systems. *International Journal on Software Tools for Technology Transfer*, 16(6):669–683, 2014.

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

[74] A. Memon, I. Banerjee, B. Nguyen, and B. Robbins. The first decade of gui ripping: Extensions, applications, and broader impacts. In *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE)*. IEEE Press, 2013.

[75] A. M. Memon. Gui testing: Pitfalls and process. *Computer*, 35(8):87–88, Aug. 2002.

[76] A. M. Memon, M. E. Pollack, and M. L. Soffa. Using a goal-driven approach to generate test cases for guis. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 257–266, New York, NY, USA, 1999. ACM.

[77] A. Mesbah, A. van Deursen, and S. Lenselink. Crawling ajax-based web applications through dynamic analysis of user interface state changes. *ACM Trans. Web*, 6(1):3:1–3:30, Mar. 2012.

[78] B. P. Miller, G. Cooksey, and F. Moore. An empirical study of the robustness of macos applications using random testing. In *Proceedings of the 1st International Workshop on Random Testing*, RT '06, pages 46–54, New York, NY, USA, 2006. ACM.

[79] B. P. Miller, G. Cooksey, and F. Moore. An empirical study of the robustness of macos applications using random testing. In *Proceedings of the 1st International Workshop on Random Testing*, RT '06, pages 46–54, New York, NY, USA, 2006. ACM.

[80] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[81] N. Mirzaei, H. Bagheri, R. Mahmood, and S. Malek. Sig-droid: Automated system input generation for android applications. In *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*, pages 461–471, Nov 2015.

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

[83] M. Mitchell, S. Forrest, and J. H. Holland. The royal road for genetic algorithms: Fitness landscapes and ga performance. In *Proceedings of the First European Conference on Artificial Life*, pages 245–254. MIT Press, 1991.

[84] I. C. Morgado and A. C. R. Paiva. The impact tool: Testing ui patterns on

mobile applications. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 876–881, Nov 2015.

[85] H. Muccini, A. Di Francesco, and P. Esposito. Software testing of mobile applications: Challenges and future research directions. In *Automation of Software Test (AST), 2012 7th International Workshop on*, pages 29–35, June 2012.

[86] H. Mühlenbein. How Genetic Algorithms Really Work: Mutation and Hill-climbing. In *PPSN*, pages 15–26, 1992.

[87] G. J. Myers. *Art of Software Testing.* John Wiley & Sons, Inc., New York, NY, USA, 1979.

[88] G. J. Myers and C. Sandler. *The Art of Software Testing.* John Wiley & Sons, 2004.

[89] B. Nguyen and A. Memon. An observe-model-exercise #x002a; paradigm to test event-driven systems with undetermined input spaces. *Software Engineering, IEEE Transactions on*, 40(3):216–234, March 2014.

[90] B. N. Nguyen, B. Robbins, I. Banerjee, and A. Memon. Guitar: an innovative tool for automated testing of gui-driven software. *Automated Software Engineering*, 21(1):65–105, 2013.

[91] C. D. Nguyen, A. Marchetto, and P. Tonella. Combining model-based and combinatorial testing for effective test case generation. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 100–110, New York, NY, USA, 2012. ACM.

[92] S. C. Ntafos. On comparisons of random, partition, and proportional partition testing. *IEEE Trans. Softw. Eng.*, 27(10):949–960, Oct. 2001.

[93] N. Nyman. Using monkey test tools. *Software Testing & Quality Engineering Magazine*, pages 18–21, 2000.

[94] A. Paiva, J. Faria, and P. Mendes. Reverse engineered formal models for gui testing. In S. Leue and P. Merino, editors, *Formal Methods for Industrial Critical Systems*, volume 4916 of *Lecture Notes in Computer Science*, pages 218–233. Springer Berlin Heidelberg, 2008.

[95] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson. Systematic mapping studies in software engineering. In *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering*, EASE'08, pages 68–77, Swinton, UK, UK, 2008. British Computer Society.

[96] H. Raffelt, M. Merten, B. Steffen, and T. Margaria. Dynamic testing via automata learning. *International Journal on Software Tools for Technology Transfer*, 11(4):307–324, 2009.

[97] V. Rastogi, Y. Chen, and W. Enck. Appsplayground: Automatic security analysis of smartphone applications. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY '13, pages 209–220, New York, NY, USA, 2013. ACM.

[98] G. J. E. Rawlins, editor. *Proceedings of the First Workshop on Foundations of Genetic Algorithms. Bloomington Campus, Indiana, USA, July 15-18 1990.* Morgan Kaufmann, 1991.

[99] A. Rosario Espada, M. del Mar Gallardo, A. Salmerón, and P. Merino. Using Model Checking to Generate Test Cases for Android Applications. *ArXiv e-prints*, Apr. 2015.

[100] M. Sama, D. S. Rosenblum, Z. Wang, and S. Elbaum. Model-based fault detection in context-aware adaptive applications. In *Proceedings of the 16th*

ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT '08/FSE-16, pages 261–271, New York, NY, USA, 2008. ACM.

[101] S. Sampath, R. Bryce, and A. Memon. A uniform representation of hybrid criteria for regression testing. *IEEE Transactions on Software Engineering*, 99(PrePrints):1, 2013.

[102] I. Satoh. *Personal Wireless Communications: IFIP-TC6 8th International Conference, PWC 2003, Venice, Italy, September 23-25, 2003. Proceedings*, chapter Testing Mobile Wireless Applications, pages 75–89. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[103] K. Sen. Effective random testing of concurrent programs. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 323–332, New York, NY, USA, 2007. ACM.

[104] E. Sherman, M. B. Dwyer, and S. Elbaum. Saturation-based testing of concurrent programs. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 53–62, New York, NY, USA, 2009. ACM.

[105] M. Srinivas and L. Patnaik. Adaptive probabilities of crossover and mutation in genetic algorithms. *Systems, Man and Cybernetics, IEEE Transactions on*, 24(4):656–667, Apr 1994.

[106] Statista. Number of apps available in leading app stores as of july 2015, 2015.

[107] T. Takala, M. Katara, and J. Harty. Experiences of system-level model-based gui testing of an android application. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pages 377–386, March 2011.

[108] W.-T. Tsai, L. Yu, F. Zhu, and R. Paul. Rapid embedded system testing using verification patterns. *Software, IEEE*, 22(4):68–75, July 2005.

[109] T. Tse and S. Yau. Testing context-sensitive middleware-based software applications. In *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*, pages 458–466 vol.1, Sept 2004.

[110] H. van der Merwe, B. van der Merwe, and W. Visser. Execution and property specifications for jpf-android. *SIGSOFT Softw. Eng. Notes*, 39(1):1–5, Feb. 2014.

[111] G. A. Wainer. *Discrete-Event Modeling and Simulation: A Practitioner's Approach*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2009.

[112] P. Wang, B. Liang, W. You, J. Li, and W. Shi. Automatic android gui traversal with high coverage. In *Proceedings of the 2014 Fourth International Conference on Communication Systems and Network Technologies*, CSNT '14, pages 1161–1166, Washington, DC, USA, 2014. IEEE Computer Society.

[113] Z. Wang, S. Elbaum, and D. S. Rosenblum. Automated generation of context-aware tests. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 406–415, Washington, DC, USA, 2007. IEEE Computer Society.

[114] H.-L. Wen, C.-H. Lin, T.-H. Hsieh, and C.-Z. Yang. Pats: A parallel gui testing framework for android applications. In *Computer Software and Appli-*

cations Conference (COMPSAC), 2015 IEEE 39th Annual, volume 2, pages 210–215, July 2015.

[115] E. J. Weyuker and B. Jeng. Analyzing partition testing strategies. *IEEE Trans. Softw. Eng.*, 17(7):703–711, July 1991.

[116] D. Whitley and K. Kauth. GENITOR: A different genetic algorithm. In *Proceedings of the 1988 Rocky Mountain Conference on Artificial Intelligence*, pages 118–130, 1988.

[117] D. Yan, S. Yang, and A. Rountev. Systematic testing for resource leaks in Android applications. In *IEEE International Symposium on Software Reliability Engineering*, pages 411–420, 2013.

[118] W. Yang, M. R. Prasad, and T. Xie. *Fundamental Approaches to Software Engineering: 16th International Conference, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, chapter A Grey-Box Approach for Automated GUI-Model Generation of Mobile Applications, pages 250–265. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[119] X. Yuan, M. B. Cohen, and A. M. Memon. Gui interaction testing: Incorporating event context. *IEEE Transactions on Software Engineering*, 37(4):559–574, 2011.

[120] O. Yurur, C. Liu, Z. Sheng, V. Leung, W. Moreno, and K. Leung. Context-awareness for mobile sensing: A survey and future directions. *Communications Surveys Tutorials, IEEE*, PP(99):1–1, 2014.

[121] R. N. Zaeem, M. R. Prasad, and S. Khurshid. Automated generation of oracles for testing user-interaction features of mobile apps. In *Software Test-*

ing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on, pages 183–192, March 2014.

[122] S. Zhang, H. Lü, and M. D. Ernst. Finding errors in multithreaded gui applications. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 243–253, New York, NY, USA, 2012. ACM.

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

[124] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou. Smartdroid: An automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, pages 93–104, New York, NY, USA, 2012. ACM.

[125] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, Dec. 1997.