FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Automatic Debugging of Android Applications

Pedro Miguel Ferreira Machado

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Rui Maranhão (PhD)

Co-Supervisor: José Campos (MSc)

17^{th} June, 2013

Automatic Debugging of Android Applications

Pedro Miguel Ferreira Machado

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: António Miguel Pontes Pimenta Monteiro (PhD)
External Examiner: José Maria Amaral Fernandes (PhD)
Supervisor: Rui Filipe Lima Maranhão de Abreu (PhD)
Co-Supervisor: José Carlos Medeiros de Campos (MSc)

17^{th} June, 2013

This work is financed by the ERDF – European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project PTDC/EIA-CCO/116796/2010.

Resumo

Nos últimos anos, o mercado dos dispositivos móveis tem crescido de forma exponencial. Este crescimento é evidenciado pelo facto de, em 2011, o número de dispositivos vendidos ter ultrapassado o número de PCs. Apesar desta evolução e das contínuas melhorias aos sistemas e arquiteturas móveis, a depuração das suas aplicações é ainda um processo manual, moroso e suscetível a erros. Embora a qualidade e fiablidade de uma aplicação possam sofrer grandes melhorias se a mesma for extensivamente testada e depurada, muitas vezes este processo não é compatível com as condições impostas pelo mercado. O diagnóstico automático de erros e/ou falhas durante o teste de *software* pode trazer grandes melhorias ao processo de depuração, ajudando assim ao desenvolvimento de aplicações com maior fiabilidade e qualidade.

A localização de falhas tem sido um grande foco de investigação, o que levou à criação de ferramentas como o Tarantula ou o GZOLTAR. Spectrum-based Fault Localization (SFL), ou localização de falhas baseada em espectros, a técnica na qual as referidas ferramentas se baseiam. Esta é uma técnica de depuração estatística que depende da informação de cobertura de código de execuções de teste. Contudo, os dispositivos móveis apresentam alguns desafios e particularidades devido à sua índole de sistema embebido e, por isso, poucos avanços têm sido alcançados na área do desenvolvimento de *software* no que diz respeito a este tipo de dispositivos.

Nesta tese é proposta uma abordagem que tem como objetivo ultrapassar os problemas causados pelas particularidades dos dispositivos móveis. Esta abordagem, denominada MZOLTAR, combina análise estática (através do `Lint`) e análise dinâmica (através do SFL) de aplicações móveis, com o objetivo de produzir relatórios de diagnóstico que ajudem a encontrar potenciais defeitos mais rapidamente. A abordagem inclui também a representação gráfica dos relatórios de diagnóstico, melhorando a perceção dos mesmos.

Para avaliar a validade e a eficácia da abordagem proposta, foi realizada uma avaliação empírica, injetando falhas em quatro aplicações Android *open-source*. Os resultados mostram que a abordagem implica um *overhead* de execução baixo (5.75% em média), enquanto que, em média, é apenas necessário inspecionar 5 componentes até que a falha seja localizada. Para além disso, é demonstrado que a informação fornecida pelo `Lint` ajuda na localização de falhas que de outra forma não seriam localizadas pelo SFL. Nesses casos, a integração com o Lint reduziu o número de componentes a inspecionar em 99.9% em média.

Abstract

In the past few years, we've been assisting to an exponential growth of the mobile devices' market. In 2011 the number of devices shipped exceeded the number of PCs. Despite this market growth and the improvements made to the mobile architectures, debugging mobile apps is still a manual, error-prone and time consuming task. While the reliability of an application can be greatly improved by extensively testing and debugging it, this process often conflicts with market conditions. Automated diagnosis of errors and/or failures detected during software testing can greatly improve the efficiency of the debugging process, thus helping to make applications more reliable.

Fault localization has been an active area of research, leading to the creation of several tools, such as Tarantula and GZOLTAR. Spectrum-based Fault Localization (SFL), the technique behind the outlined tools, is a statistical debugging technique that relies on code coverage information. However, the embedded nature of mobile devices poses some particular challenges, thus very few has been reported in the area of mobile software.

This thesis proposes an approach to overcome the challenges presented by the mobile devices architecture. This approach, dubbed MZOLTAR, that combines static (using `Lint`) and dynamic analysis (using SFL) of mobile apps to produce a diagnostic report to help identify potential defects quickly. The approach also offers a graphical representation of the diagnostic report, making it easier to understand.

To assess the validity and performance of MZOLTAR, an empirical evaluation was performed, by injecting faults into 4 real open-source Android applications. The results show that the approach requires low runtime overhead (5.75% on average), while the tester needs to inspect 5 components on average to find the fault. Furthermore, it demonstrates that Lint helps revealing bugs that otherwise would go undetected by the SFL fault localization technique. In those cases, the integration with Lint reduced the number of components to inspect by 99.9% on average.

Acknowledgements

The journey to write this thesis would have not been possible without the people that stood by me during all these years. First of all, I would like to thank my supervisors Rui Maranhão and José Campos for their help and for making me believe I was up to the challenge. Their motivation was a key point as they managed to transform an area I had never considered before into one that I am passionate about. Special thanks to my dear friend Alexandre Perez, for all the times I bored him with my problems and for all the help he gave me. Thanks to my lab partners, Carlos Gouveia, Luís Pinho, João Batista and José Barbosa for helping me whenever I needed and for the fun times. I would also like to thank Nuno Cardoso for all the helpful ideas and feedback he gave me throughout this thesis (and for the beer, the beer helped a lot!). I am also grateful to Dr. Arjan J.C. van Gemund for his valuable and helpful feedback.

To my dear friend João Santos, even though he is far, I would like to thank for all the support and fun times we had chatting and playing along with Luís Silva. To my friends Fernando Fernandes, Filipe Carvalho and Mariana Ponce who have always been there for me.

I would like to thank my girlfriend, Margarida Fonseca, for all her support and understanding, for believing in my potential and for all the times she stood by my side when I had to work. I love you!

Finally, I would like to address the biggest thank of them all to my family, specially to my parents Luísa Machado and Adérito Machado, and grandparents Fernanda Ribeiro and Domingos Ferreira, for giving me the chance to make them proud and to repay all they have done for me. All the sacrifices, all the times I had to stay up late, all the times I couldn't be with you, it pays off when I get to see the pride stamped in your faces. Thank you!

Pedro Miguel Ferreira Machado

"Quality is the ally of schedule and cost, not their adversary. If we have to sacrifice quality to meet schedule, it's because we are doing the job wrong from the very beginning."

James A. Ward

Contents

List of Figures

List of Tables

Abbreviations

ADB Android Debugging Bridge

ADT Android Development Tools

.dex Dalvik Executable

DDMS Dalvik Debug Monitor Server

DVM Dalvik Virtual Machine

GROPG A Graphical On-Phone Debugger

GUI Graphical User Interface

IDE Integrated Development Environment

JDWP Java Debug Wire Protocol

JVM Java Virtual Machine

LOC Line Of Code

NDK Native Development Kit

SDK Software Development Kit

SFL Spectrum-based Fault Localization

SUT System Under Test

UI User Interface

VM Virtual Machine

Chapter 1

Introduction

In the past years, the market of mobile devices such as smartphones and tablets has been growing exponentially. The number of shipped mobile devices even exceeded the PCs in 2011. The two main operating systems in the market, Apple's iOS and Google's Android, have been competing for about 5 years, and in the first quarter of 2013 Android had 75% of the market share while Apple had 17%[1]. This has the potential to represent a big paradigm shift in some technological areas and implies substantial changes and adjustments in the software development area.

The process used to develop software for mobile devices is the same used to develop any other software. Generally the process follows four main phases: design, implementation, testing and release (see Figure 1.1).

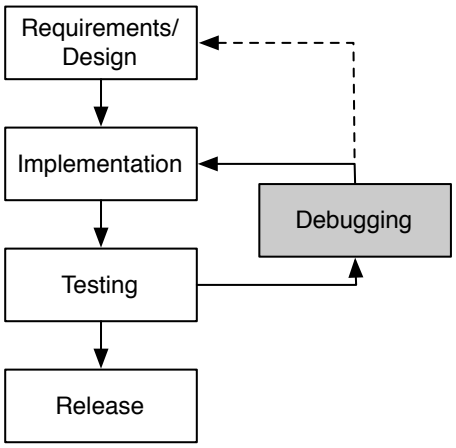


Figure 1.1: Software development process.

However, it is highly probable that not all tests pass at the first try and there may be needed some adjustments in the implementation. This implies adding a new phase to the process, that includes changing the implementation (although the focus of this thesis is the implementation, in some cases the design itself may need adjustments) to assure the application is functioning

[1]IDC Worldwide Quarterly Mobile Phone Tracker - `http://www.idc.com/getdoc.jsp?containerId=prUS24108913`, 2013.

correctly, according to its specification. Therefore a cycle is created, following a debugging phase whenever a test or multiple tests fail. This phase includes correcting the implementation to mitigate the problem, and after the adjustments, the application is tested again, until all tests pass and all the conditions for successful release are met. Debugging is not easily estimated and, besides being the most error-prone phase in the software development life cycle, it consumes a great amount of time and other resources. Given those facts it is important to drastically reduce the debugging costs to achieve better software quality with less resources [Tas02, HS02].

Although several debugging tools are available, not much has been reported in the area of mobile software, therefore debugging mobile applications is still a manual and time-consuming task. This thesis' main goal is to ease the task of debugging mobile applications, increasing reliability and quality, while reducing the usual time-to-market.

1.1 Concepts and Definitions

Throughout this thesis, the following terminology is used [ALRL04]:

- a *failure* is an event that occurs when delivered service deviates from correct service
- an *error* is a system state that may cause a failure
- a *fault* (defect/bug) is the cause of an error in the *system*.

In this thesis, this terminology is applied to software programs, where faults are bugs in the program code. Failures and errors are symptoms caused by faults in the program. The purpose of fault localization is to pinpoint the root cause of observed symptoms (failures).

Definition 1 *A software program* Π *(a mobile app in the context of this thesis) is formed by a sequence M of one or more statements.*

Definition 2 *A test suite* $T = \{t_1, \ldots, t_N\}$ *is a collection of test cases that are intended to test whether the program follows the specified set of requirements. The cardinality of* T *is the number of test cases in the set* $|T| = N$.

Definition 3 *A test case t is a* (i, o) *tuple, where i is a collection of input settings or variables for determining whether a software system works as expected or not, and o is the expected output. If* $\Pi(i) = o$ *the test case passes, otherwise fails.*

1.2 Motivation

Despite recent advances in the mobile software area, the development of applications for mobile devices - such as Android apps - still poses interesting challenges [EMK13]:

- The large amount of available devices and their large range of specifications (e.g., cpu speed, screen resolution), makes it difficult to check the consistency and ensure portability between different devices and platforms;
- Testing apps for each target platform requires the development of several versions. Furthermore, the available testing frameworks have serious limitations for testing mobile specific features;
- Developers claim that better analysis tools and techniques to help debugging apps are seriously needed. In fact in the World Quality Report [CCSH12], ⅔ of surveyed developers mentioned that they do not have the proper tools to test and debug mobile apps, despite the available tools such as provided by Android Software Development Kit (SDK) and Android Development Tools (ADT) plugin.

Software reliability can generally be improved through extensive testing and debugging, however this often conflicts with market conditions. Often, software cannot be tested exhaustively, and of the bugs that are found, only those with the highest impact on the user-perceived reliability can be solved before the release. Therefore, applications tend to be released with bugs that are not easily perceived but can cause problems in the future. Given the low impact of the most software bugs in this area, the reliability is brought down to a commercial acceptable level. The goal is not to loose out to the competitors and achieve a acceptable product, with a perceivable value to the costumer, in spite of its imperfections.

In this typical scenario, testing reveals more bugs than can be solved, and debugging is a bottleneck for improving reliability. Although the mobile applications market has been growing a lot in the past few years, and even with the fast evolution of the mobile architectures we have been assisting, debugging mobile applications is still a manual and time-consuming task. Automated debugging techniques, based on data gathered from program executions [WDLG12, HZZ+09, AZGvG09, LFY+06, WWQZ08, SSPC13, SF12] or on expected program behaviour [MS03, MS08, dK09] may be used to reduce this bottleneck.

Locating a fault is an important step in actually fixing it. SFL is a technique, which is amongst the best performing techniques, that helps identifying the root cause of observed failures, relying on program execution data and test pass/fail information. GZOLTAR[2] [CRPA12], which focuses on Java programs, and Tarantula [JH05], which focuses on C programs are examples of tools offering the SFL technique. Since it is lightweight, SFL has been successfully applied in the context of embedded software [ZAGvG07, ZPA+08]. Despite these tools and the increasingly active research in the area of fault localization, not much has been reported in the area of mobile software.

Furthermore, given the particularities of mobile apps development, some faults are of a more static nature (e.g., not declaring an activity in the *manifest* file[3]). As this may *conceal* the fault, SFL simply misses it. Thus, decreasing considerably the quality of the diagnostic report. `Lint`[4]

[2] GZOLTAR homepage `http://gzoltar.com`, 2013.

[3] The manifest presents essential information about the application to the Android operating system; information the system must have before it can run any of the application's code.

[4] `Lint` homepage `http://developer.android.com/tools/help/lint.html`, 2013.

is a static analysis tool that was adapted to Android, being distributed with the Android SDK. It scans project sources seeking for potential static defects.

Tools like GZOLTAR provide visual representations of the diagnostic report. These visualizations are a *translation* of the report into an intuitive representation that ease and speed up the fault localization process.

Hereupon, the application of SFL in conjunction with the information provided by `Lint` checks would ease the debugging of mobile apps. Moreover, combining the two analysis and reflecting this combination in any provided visualizations would positively affect the fault localization process.

1.3 Main Goals

Given the main limitations of the mobile devices, their embedded nature and the fact that they are resource constrained devices, some research questions arise:

- Is the MZOLTAR's instrumentation overhead negligible?
- Does MZOLTAR yield accurate diagnostic reports under Android device's constrained environment?
- Does the integration with `Lint` contribute to a better diagnostic quality?

To answer to these questions, the purpose of this thesis was to develop a toolset that relies on SFL and `Lint`, devising a way to combine both dynamic and static analysis. Furthermore, the developed toolset provides visual representations of the diagnostic report(visualizations) that ease the debugging process. These visualizations will present, clearly, the combination between static and dynamic analysis. Finally, an empirical evaluation to assess MZOLTAR's performance and verify its applicability to the context of mobile apps was performed.

This way, the developed toolset mitigates some time and reliability constraints faced by developers, increasing the quality of the developed and released mobile applications.

1.4 Document Structure

Besides Chapter 1, the Introduction, this document is composed by six more chapters.

Chapter 2 describes the Android platform's history and architecture.

Chapter 3 aims to present the related work in the mobile applications debugging field. It also presents some automatic fault localization tools and techniques.

Chapter 4 contains the description of the proposed solutions as well as the tools this solutions are based on.

Chapter 5 details the developed toolset as well as its underlying technique.

Chapter 6 presents the outcome of the performed empirical study.

Chapter 7 presents the conclusions drawn from this thesis' work.

Chapter 2

Android Operating System

Android is an open source Linux-based operating system targeted for mobile or embedded devices. Typically, Android applications are developed in Java language. However, languages such as C or/and C++ may also be used to compile into native code. Note, that Android reuses the Java language syntax and semantics, but it does not provide the full class libraries and APIs bundled with Java SE. Java development is mainly supported by a comprehensive set of development tools SDK, while native code is supported by the Native Development Kit (NDK). Android applications have a specific lifecycle that differs from the Java regular applications' lifecycle. Instead of a main function, Android applications' main components are: *Activities*; *Services*; *Content providers*; *Broadcast receivers*. Furthermore, Android applications are not only comprised of source files, but also include resource files (mainly related to the specification of layouts and translations) and a manifest file (as mentioned before, responsible to provide the necessary information about the application to behave correctly to the Android system).

2.1 History

Android's alpha and beta versions where launched in late 2007. Its first commercial version, Android 1.0, was released in 2008. The first Android device, HTC Dream, incorporated this version of the operating system. So far seventeen versions of the API have been released, the most recent being Android 4.2 also known by the code-name *Jelly Bean*.

As stated by Google, the company who leads the Android open-source project, "Android is an open-source software stack for mobile phones and other devices. (...) The goal of the Android Open Source Project is to create a successful real-world product that improves the mobile experience for end users.". It is open-source nature and permissive licensing allows manufacturers to freely modify and distribute the operating system with their devices, therefore allowing some flexibility in what it touches to compatibility issues. As the years passed, Android also gathered a big community of developers who willingly modify and even add features to the operating system in

order to improve the performance of some devices. There is also a big community of Android applications developers, hence Google Play application market as reached the 25 billion downloaded applications and currently has around 700 thousand applications available to download.

In the first quarter of 2013 Android had a market share of 75.0% and according to the its official Engineering team it reached the 500 million devices activated worldwide and about 1.3 million activations per day. On the other hand, Android's competitors registered smaller shares: Apple's iOS had a 17.3% while Blackberry, Linux, Symbian and Windows Phone had about 7.7% all together.

2.2 Android Manifest

The manifest file (AndroidManifest.xml) must be present in the root directory of every application. The Android system relies on this file to access file about the application before being able to successfully run it (i.e., if an activity is not declared the system will not be able to run it, if the internet permission is not declared the system will not be able to perform http connections). The manifest is responsible for:

- Declaring an unique identifier for the application (package name);
- Describing application components (see Subsection 2.3);
- Determining which processes will host application components;
- Declaring which permissions the application must have to access protected parts of the API and interact with other applications;
- Declaring the permissions that others are required to have to interact with the application's components;
- Listing the Instrumentation classes that provide profiling and other information as the application is running;
- Declaring the minimum level of the Android API that the application requires;
- Listing the libraries that the application must be linked against;

2.3 Components

As stated before, Android applications have a very specific lifecycle. Their main components are: *Activities*; *Services*; *Content providers*; *Broadcast receivers*.

An **Activity**[1] is the most common application component that provides a graphical user interface with which users can interact to perform actions. Examples of Android activities are "dial a

[1] Android Activities homepage `http://developer.android.com/guide/components/activities.html`, 2013.

phone number", "take a photo", "send an email", "view a map", etc. Activities are implemented as a subclass of the Android SDK's Activity class and can be customized to perform any action requiring user interaction. The lifecycle of an activity (see Figure 2.1) is based on a set of states and the actions performed when the activity changes its state.

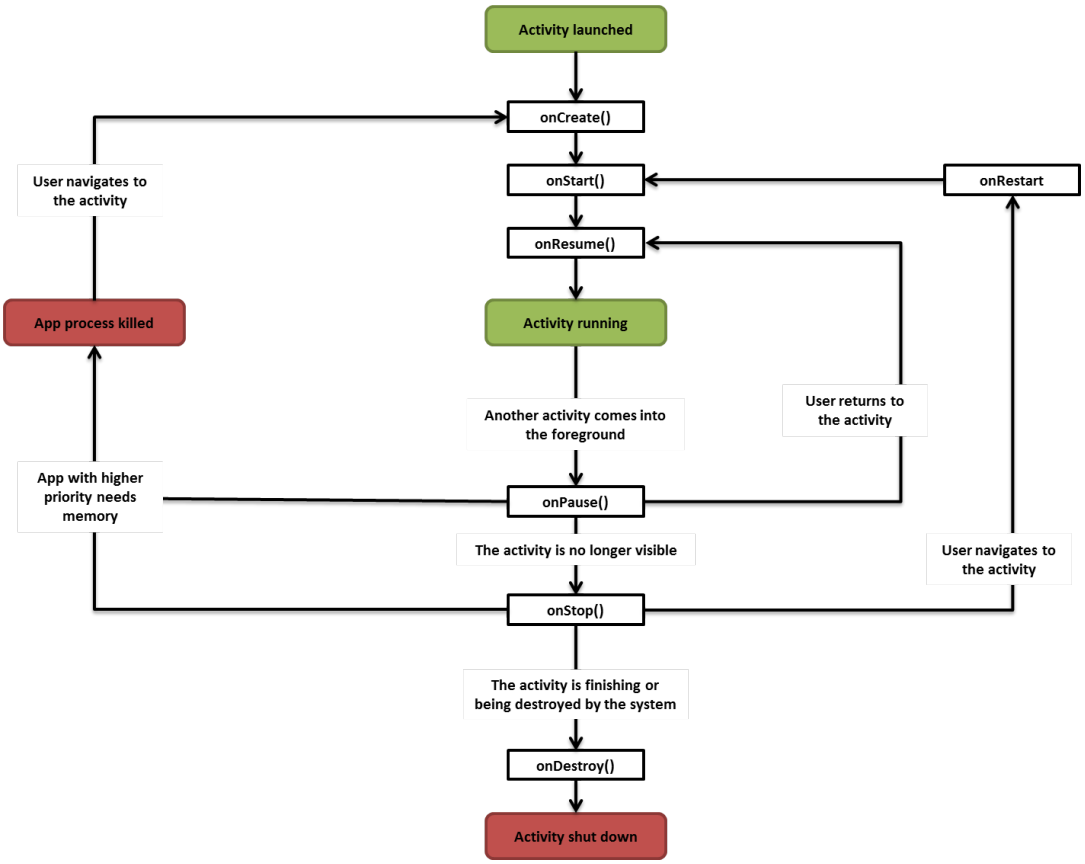


Figure 2.1: Android activity lifecycle diagram

onCreate() - Called when the activity is first created. Most activity initialization should be made here, such as UI inflation, data binding or recovery of a previously frozen state. It's always followed by onStart();

onRestart() - Called after if onStop() was called before the activity starts again. Always followed by onStart();

onStart() - Called when the activity becomes visible to the user. Followed by onResume() if the activity comes to the foreground or onStop() if it becomes hidden;

onResume() - Called when the activity will start interacting with the user. Actions such as animations or opening exclusive access-devices (e.g., camera) should be done here. Always followed by onPause();

onPause() - Called when a new activity is launched in front of the previous activity. Persistent data should be saved here. The actions made here must be quick and simple, because the

new activity will only start when onPause() returns. In most cases this activity is followed by onStop();

onStop() - Called when the activity is no longer visible to the user. This method may only be called when the system does not have enough memory to keep the activity running. It can be followed by onRestart() if the activity becomes visible to the user again, or by onDestroy() if the activity is going to be destroyed by the system;

onDestroy() - Called when the activity is destroyed either by the system or by an explicit call to the *finish* method.

The state may change when events are triggered. For instance, when an Activity is first created it will be on the *Created* state, and then its state will be changed to *Started*. Between any state transition, an action is executed and it can be customized by overriding each of the callback methods in the Activity class. When the activity is created and before setting its state to *Created*, the *onCreate()* method will be called and the consequent action will be performed, as well as the *onStart()* method will be called during the transition between the *Created* and *Started* states.

A **Service**[2] is an application component intended to perform long-running operations, usually in the background. Examples of services are playing music, perform network operations or to supply functionality for other applications to use. Unlike activities, services do not provide a user interface.

Content providers[3] are components that manage access to a structured set of data, as they encapsulate the data and provide mechanisms to define data security. A content provider is the alternative to an SQLiteDatabase and should be used when multiple applications use the stored data. The encapsulated data is provided to applications through a ContentResolver[4] interface. Android itself includes content providers that manage data such as audio, video, images, and personal contact information.

Broadcast Receivers[5] respond to system-wide announcements, such as a broadcast announcing that the screen has turned off, the battery is low, or a picture was captured. Commonly, a broadcast receiver is a *gateway* to other components and should do a very small amount of work. For instance, it might initiate or an activity a service to perform some work based on the event.

[2]Android Services homepage `http://developer.android.com/guide/components/services.html`, 2013.

[3]Android Content Providers homepage `http://developer.android.com/guide/topics/providers/content-providers.html`, 2013.

[4]Android Content Resolver homepage `http://developer.android.com/reference/android/content/ContentResolver.html`, 2013.

[5]Android Broadcast Receivers homepage `http://developer.android.com/reference/android/content/BroadcastReceiver.html`, 2013.

2.4 System Architecture

Android is a stack of different layers which includes an operating system, middleware and key applications. Each layer provides different services to the layers just above it. The architecture layers are specified in Figure 2.2.

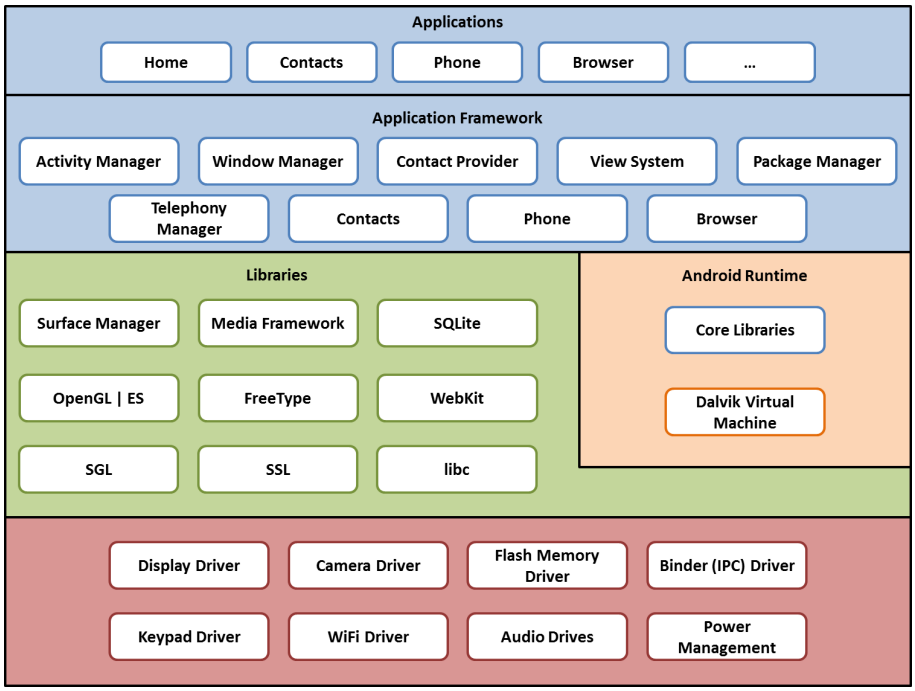


Figure 2.2: Android system architecture

2.4.1 Applications and applications framework

Android base releases have a set of core applications that are indispensable for the systems proper functioning. Additionally the development framework used to develop the core applications is provided as an open development platform. It allows the developers to take advantage of the device's hardware and features to their applications benefit.

Although Android applications are written mainly in Java, as referred, there are some points in which Android applications and standard Java applications differ from each other. The differing points are the following:

- Android applications have a specific lifecycle that differs from the Java regular applications' lifecycle. Instead of having a main function, Android applications are based on activities and each activity transitions between different states in its lifecycle (see Figure 2.1). In order to perform some actions in the state transitions (depending on the needs actions may be performed in different state transitions) the corresponding functions must be overridden.
- Some libraries of JavaSE and Android may not be compatible as they may not include all JavaSE elements or are exclusive to Android (see Section 2.4.3).

- A Java regular application uses Java Virtual Machine (JVM) while Android uses Dalvik Virtual Machine (DVM) (see Section 2.4.3).
- Android uses XML to declare its User Interface (UI) elements while regular Java applications use platforms like SWING or JavaFX.

2.4.2 Libraries

Android includes a set of C/C++ libraries used by various components of the Android system. Developers can access these capabilities through the provided development framework. These capabilities include system libraries, media libraries, graphic libraries, browser, database engine. The libraries are hardware specific. There follow some examples of core libraries of the Android system:

- Surface Manager: composes UI on the screen;
- SGL: the underlying 2D graphics engine;
- OpenGL ES: 3D library implemented based on OpenGL ES 1.0 APIs;
- Media Framework: supports playbacks and recording of various audio, video and picture formats;
- Free Type: bitmap and vector font rendering;
- WebKit: browser engine which powers both the Android browser and an embeddable web view;
- libc: implementation of the standard C system library (libc), tuned for embedded Linux-based devices;
- SQLite: relational database engine available to all applications.

2.4.3 Android Runtime

Android Runtime consists of DVM and Core Java libraries. DVM runs Dalvik Executable (.dex) files unlike the JVM that runs .class files. But Java source files are not directly compiled into .dex files. They first are compiled into .class (java bytecode) files and then transformed into .dex files (dalvik bytecode). Dalvik Executable files are optimized to achieve higher efficiency on resource constrained environments like mobile devices. Dalvik was written to allow the running of multiple virtual machines efficiently and, therefore, allow each application to run on its own virtual machine instance, then providing security, isolation, multi-level memory management and threading support.

It is also important to highlight the Android permission mechanism, that enforces restrictions on the specific operations that a particular process can perform or grant access to specific pieces

of data. Android applications have no permissions associated by default, which prevents the application from doing anything that would adversely impact the user experience or any data on the device. If any feature uses any method that requires a given permission, that permission must be explicitly specified in the manifest by the developer. Minimizing the required permissions by declaring just the ones the application really needs is a must.

Some of the Java core libraries may not be exactly equal to the JavaSE libraries as they may not include all of the elements of their JavaSE equivalents. For instance the method isEmpty() included in the java.lang.String class since the 1.5 version of JavaSE launched in 2004 that was only included in version 9 of Android API. Other example is the System.out and System.err streams, which do not provide any output when used in Android, being the use of the android.util.Log class encouraged. Oh the other hand some APIs are exclusive to Android, such as the contacts API.

2.4.4 Linux Kernel

The Linux Kernel it is the lowest layer and acts as an abstraction layer between the hardware and all the architectural layers. There Kernel never interacts directly with the user. Its importance stems from the fact that it provides the following functions in the Android system:

- Hardware Abstraction
- Memory Management Programs
- Security Settings
- Power Management Software
- Other Hardware Drivers
- Support for Shared Libraries Network Stack

Chapter 3

Related Work

When analysing debugging tools, there are two categories that can be taken into account:

- **manual debugging tools** based on a step-by-step execution
- **automatic debugging tools** on the history of several executions or on expected program behaviour

Debugging Android applications is a manual and not trivial task. Moreover it is a rather time-consuming and error-prone process. Android SDK provides several tools for this purpose, all of them fitting in the **manual debugging tools** category.

Statistical debugging is type of automated debugging and has been a very active subject of research in the past years. Some automatic fault localization tools like GZOLTAR or Tarantula have been developed, using a SFL approach (see Subsection 3.3.1), and have been proven to be accurate and efficient. Nevertheless, very few has been reported in the area of mobile software.

Despite the myriad of techniques and approaches, there are still shortcomings when applying these techniques in the context of mobile, resource-constrained apps. Available automated fault localization toolsets do not offer easy integration into the mobile apps world. As a consequence, manual approaches are still prevalent in the mobile apps debugging and testing phases, and the debugging tools available for mobile apps only offer manual debugging features [EMK13].

In this chapter, the Android testing framework will be presented along with some debugging tools provided by the Android SDK, as well as some automatic approaches.

3.1 Testing Android Applications

The testing framework provided by Android SDK extends JUnit[1] with specific features that provide access to Android system components and ease the implementation of several testing strategies [Mil11].

[1] JUnit homepage `http://www.junit.org/`, 2013.

Android testing framework also provides an instrumentation framework to control the tested application. With the instrumentation framework is possible to inject mock Android system objects to simulate specific situations.

Test projects are similar to the main applications' projects. This way the user does not need to learn a new set of tools and can easily design and build tests for the developing application.

Figure 3.1 summarizes the Android testing framework. In this section the instrumentation framework will be analysed, along with MonkeyRunner, Monkey and Robotium tools.

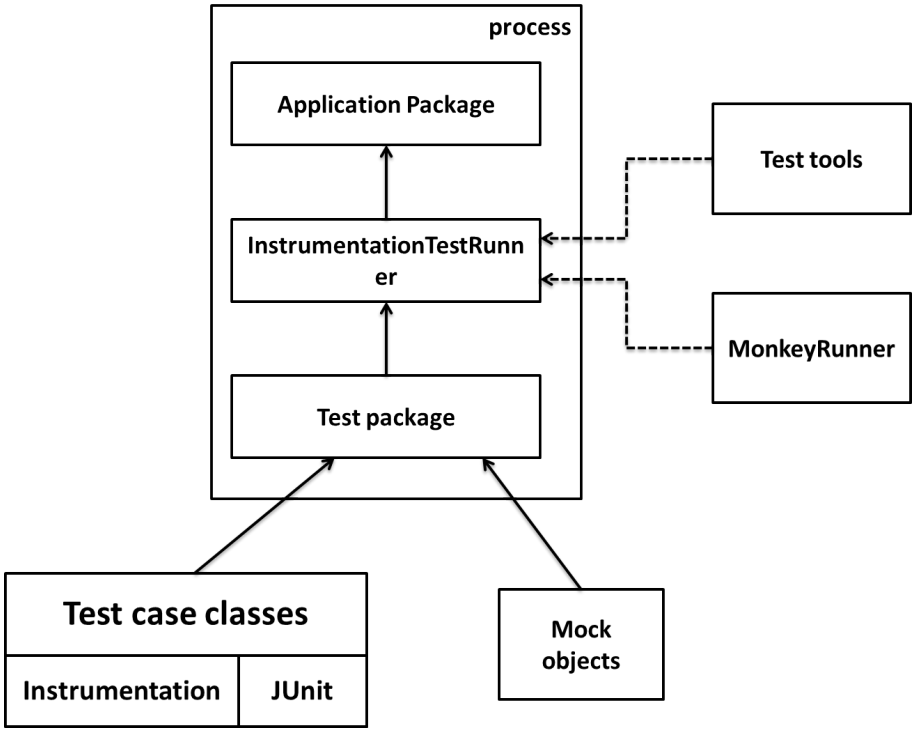


Figure 3.1: Android test framework diagram.

3.1.1 Instrumentation framework

Android instrumentation is a set of control methods or "hooks" in the Android system. With instrumentation is possible to invoke callback methods that are exclusively controlled by the system and cannot be invoked directly, providing a way to, for example, run through the lifecycle of a component step by step.

Instrumentation also controls the way tests run, by shutting down any previous running instances of the main application on the device and controlling the test runner responsible for running the tests.

3.1.2 MonkeyRunner

MonkeyRunner is a tool that provides an API to write Python programs that control a device or emulator from outside the application code. The programs run on the developing system and not on the device itself, sending specific events to the device.

The results are presented as UI screenshots and can even be compared to a set of screenshots that are known to be correct.

3.1.3 Monkey

The infinite monkey theorem states that a monkey hitting keys at random on a typewriter keyboard for an infinite amount of time will almost surely type a given text, such as the complete works of William Shakespeare. Monkey is a tool that transposes this theorem to the Android applications' context. In this context an entity that produces random events on the device could crash the application under test.

Monkey is a command line tool and runs on a device or emulator and generates pseudo-random streams of user events such as clicks, touches, or gestures, as well as a number of system-level events. The tool provides the possibility of performing stress-tests to an application, by using several configurations like event types and frequencies.

3.1.4 Robotium

Robotium[2] is a test framework created to make it easy to write automatic black-box test cases for Android applications. It simulates touching, clicks, typing, and other UI actions relevant for Android applications. The presentation of the results is very similar to the JUnit results presentation (red/green bar to represent failed/passed tests).

To write and run tests with Robotium, no source code is required. With just the apk and minimal knowledge of the application under test it is possible to write solid test cases. The tests are written to follow a given UI flow and assert the properties of some system components.

3.2 Debugging of Android Applications

As stated, Android SDK provides a set of tools that make it possible to debug the developed applications. In this section each one of these tools will be presented and briefly described.

3.2.1 LogCat

The Android logging system, dubbed LogCat, is a way to output information directly from the device and track application events (see Figure 3.2). It can be viewed both from command line through Android Debugging Bridge (ADB) or from Eclipse (or some other external tool that implement an interface to ADB). The use of JavaSE system outputs, System.out and System.err, is

[2]Robotium homepage `http://code.google.com/p/robotium/`, 2013.

discouraged in Android, as they are redirected to LogCat (or simply does not display anything in some old versions of Android).

LogCat has several fields and can be displayed in some kinds of formats, depending on its main goal. LogCat fields are the following:

- Time - The time at which the LogCat message was generated in the device
- Priority - Indicates the severity of the message
- PID (Process ID) - ID of the process that generated the message
- TID (Task ID) - ID of the task that generated the message
- Tag - Message tag
- Message - Message content itself

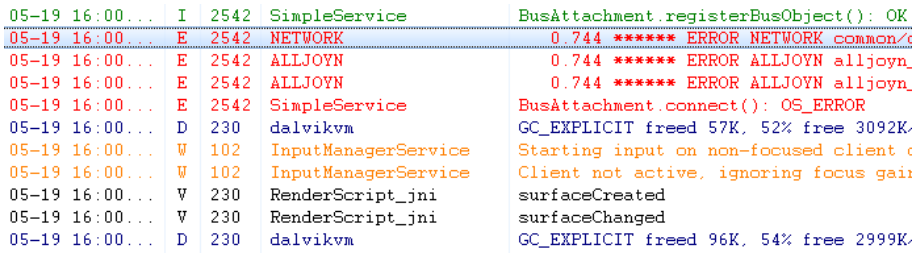


Figure 3.2: LogCat output example

Not only the messages originated in the applications under development are shown, as the Android system itself often outputs its own LogCat messages, in most cases, related with system events. Many types of information is output in the LogCat system and the severity of each message may differ, thus messages can have different priorities. The possible severities for a LogCat message are the following:

- V — Verbose (lowest priority)
- D — Debug
- I — Info
- W — Warning
- E — Error
- F — Fatal
- S — Silent (highest priority, on which nothing is ever printed)

Although LogCat is not the most precise debugging tool, as an output system it lets users track specific behaviours or states of an application, thus providing a versatile way for tracking what the application is doing in a specific situation.

3.2.2 Android Debugging Bridge (ADB)

ADB is a command line tool included in the Android SDK that plays a very important role in the process of debugging Android applications. It represents a direct connection layer between a device and the developing system.

ADB acts as a intermediary between a device and the developing system, not only providing a tool for managing the devices connected (install/remove applications, sync files) and run a UNIX shell in a given device but also serving as a connection between the device and the developing system itself.

An ADB device daemon runs on the device, while a ADB host daemon runs on the developing system, providing a debugger that connects to this interface a way to collect information to successfully debug an application running on a device or emulator.

3.2.3 Dalvik Debug Monitor Server (DDMS)

DDMS is a graphic debugging tool (see Figure 3.3), both accessible through Eclipse or command line, whose features include:

- Viewing heap usage for a process
- Tracking memory allocation of objects
- Working with an emulator or device's file system
- Examining thread information
- Starting method profiling
- Using the Network Traffic tool
- Using LogCat
- Emulating phone operations and location
- Changing network state, speed, and latency
- Spoofing calls or SMS text messages
- Setting the location of the phone

When it starts, DDMS connects to ADB, then creating a Virtual Machine (VM) monitoring system that notifies DDMS whenever a device is connected or disconnected. As in Android every application runs in its own process, each of which running in its own VM, DDMS opens a connection to each VM's debugger through the ADB daemon on the device and then assigns a debugging port to each VM. When a debugger connects to one of these ports, all traffic from the associated VM is forward to that port.

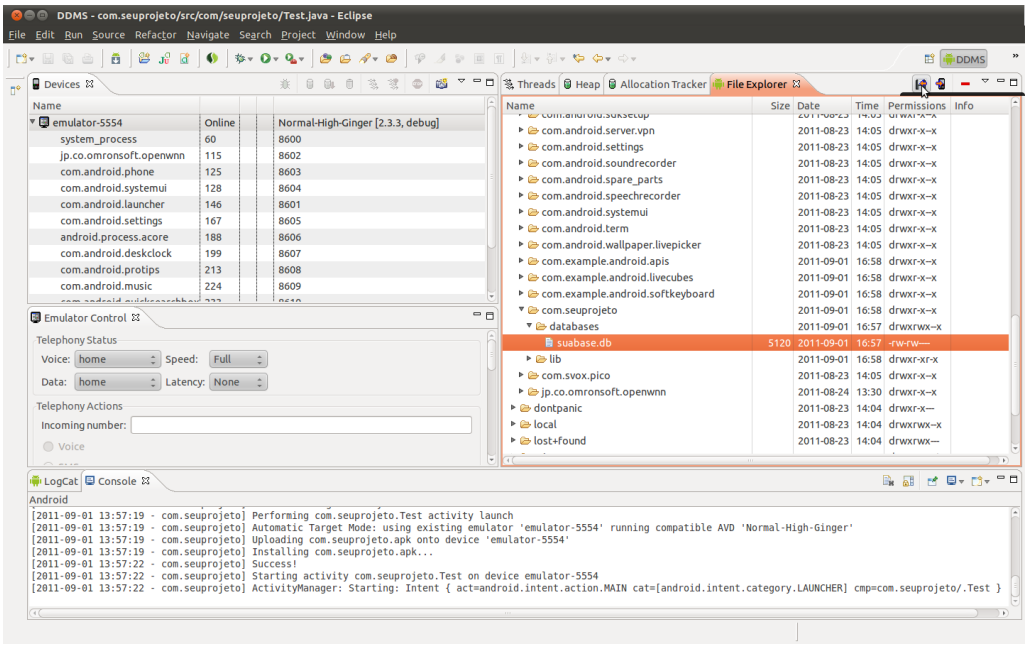


Figure 3.3: DDMS screenshot

3.2.4 Java Debug Wire Protocol (JDWP) debugger

JDWP is a protocol used for communication between a debugger and a JVM. 1The DVM supports the JDWP protocol to allow debuggers to attach to a VM. Each application runs in a VM and exposes a unique port to which a debugger can be attached to via DDMS. If the goal is to debug multiple applications, attaching to each port might become tedious, so DDMS provides a port forwarding feature that can forward a specific VM's debugging port to port 8700. It is possible to switch freely from application to application by highlighting it in the Devices tab of DDMS. DDMS forwards the appropriate port to port 8700. Most modern Java IDEs include a JDWP debugger, but a command line debugger, such as jdb, can also be used. Figure 3.4 shows how the various debugging tools work together in a typical debugging environment.

3.2.5 Traceview

A trace graphical viewer that shows trace file data for method calls and times for the developing application, giving useful information that helps profiling its performance. A trace log file, used by traceview, can be generated either by adding tracing code to the application or by DDMS. After the files are loaded, two panels are then provided:

- **Timeline Panel** (see Figure 3.5) which describes visually when each thread and method started and stopped. Each method is shown in another colour (colours are reused in a round-robin fashion starting with the methods that have the most inclusive time). The thin lines underneath the first row show the extent (entry to exit) of all the calls to the selected method.

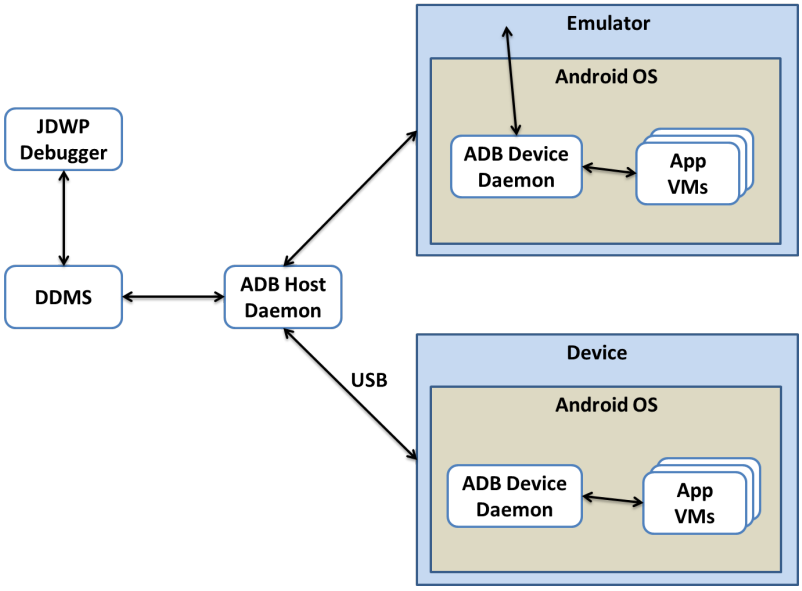


Figure 3.4: The architecture

- **Profile Panel** (see Figure 3.6) presents a summary of the time spent in the method. Both the exclusive time (time spent in the method) and inclusive time (time spent in functions called by the method) are presented, as well as the percentage of total time. This panel also presents the number of calls to this method plus the number of recursive calls in the last column.

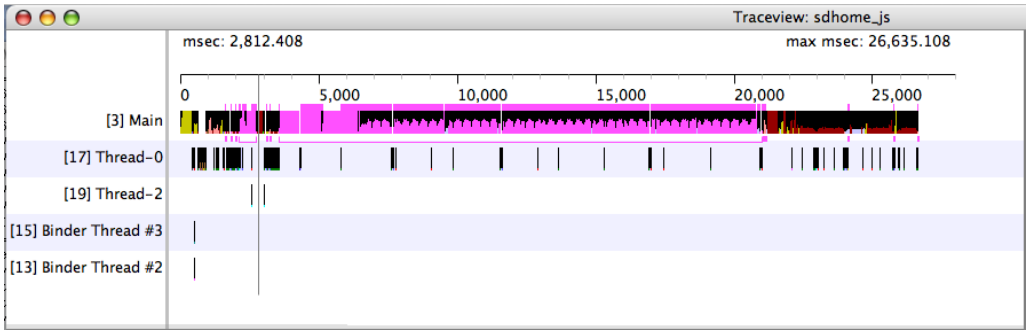


Figure 3.5: The timeline panel of the traceview tool

Name	Incl %	Inclusive	Excl %	Exclusive	Calls+Rec
4 android/webkit/LoadListener.nativeFinished ()V	66.6%	17734.382	53.2%	14161.950	14+0
3 android/webkit/LoadListener.tearDown ()V	100.0%	17734.382			14/14
6 android/view/View.invalidate (IIII)V	19.8%	3516.410			2413/2853
57 android/webkit/BrowserFrame.startLoadingResource (ILjava	0.3%	44.636			3/15
53 java/util/HashMap.put (Ljava/lang/Object;Ljava/lang/Objec	0.0%	6.223			6/326
20 android/webkit/JWebCoreJavaBridge.setSharedTimer (J)V	0.0%	2.593			2/730
378 android/view/ViewGroup.requestLayout ()V	0.0%	1.139			2/54
315 java/util/HashMap.<init> (I)V	0.0%	0.879			3/41
629 android/webkit/BrowserFrame.loadCompleted ()V	0.0%	0.285			1/1
598 android/webkit/WebView.didFirstLayout ()V	0.0%	0.231			1/2
703 android/webkit/BrowserFrame.windowObjectCleared (I)V	0.0%	0.036			1/2
5 android/webkit/JWebCoreJavaBridge$TimerHandler.handleMessa	16.3%	4342.697	0.5%	132.018	730+0
6 android/view/View.invalidate (IIII)V	15.6%	4161.341	1.2%	319.164	2853+0

Figure 3.6: The profile panel of the traceview tool

3.2.6 HierarchyViewer

Graphical programs that makes it possible to debug and profile an application's UI, by providing a visual representation of the layout's View hierarchy (the View Hierarchy window) and a magnified view of the display (the Pixel Perfect window).

The two panels provided are detailed next:

- **View Hierarchy window** (see Figure 3.7) displays the View objects that form the UI of the Activity that is running on the device or emulator. It can be used to look at individual View objects within the context of the entire View tree. For each View object, the View Hierarchy window also displays rendering performance data. It is also possible to observe an individual View from the View tree in detail, obtaining more detailed information about it, thus easing the debugging of a specific part of the UI.

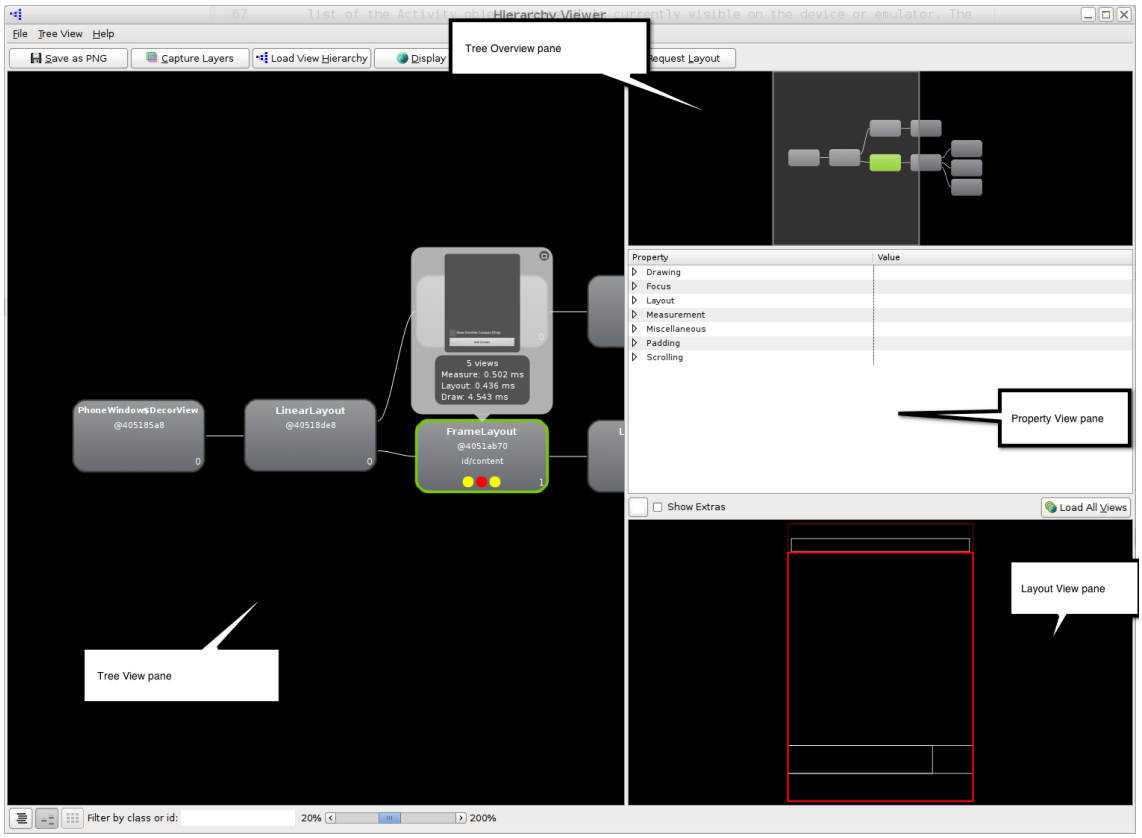


Figure 3.7: HierarchyViewer - View Hierarchy window

- **Pixel Perfect window** displays a magnified image of the screen that is currently visible on the emulator or device. In this window, it is possible to examine the properties of individual pixels in the screen image. It is also possible to use the Pixel Perfect window to help lay out the application UI based on a bitmap design.

3.2.7 Lint

`Lint`[3] is a static analysis tool for Android applications. It was introduced in the version number 16 of the Android SDK Tools. It is based on an homonym tool developed at Bell Labs[4] in 1977 that flags some suspicious and non-portable constructs (likely bugs) in C language source code. Android's `Lint` acts on the Java source code used in Android applications and also aims at flagging potential bugs and optimization improvements for correctness, security, performance, usability, accessibility, and internationalization. In addition, Android's `Lint` also analyses Android specific resources, like layout files or the Android manifest file, to point out problems specific to the platform.

When analysing the source code and the resources of an Android project, `Lint` performs a series of checks that are associated with specific known issues. Each issue is characterized by three properties

Category
: Indicates the nature of the issue, such as *Usability*, *Correctness* or *Performance*. A subcategory may also be presented, e.g., *Usability:Icons*;

Severity
: Indicates the impact the issue may represent to the application. There are three levels of severity: *Warning* (lowest threat), *Error* and *Fatal* (highest threat).

Priority
: Indicates the priority of the issue. If an issue has a priority of 10 it should be accessed before another issue with priority 6. The priority assumes values between 1 and 10.

`Lint` generates a report with all the errors found after analysing the code. `Lint`'s proprieties can be customised, to ignore some issues that are not relevant in a project, to change the severity of a specific problem or to add new/customised checks.

3.2.8 A Graphical On-Phone Debugger (GROPG)

Besides the tools provided by the Android SDK, there is another tool that presents a different approach. GROPG[5] [NCT13] was developed at University of Texas at Arlington and is a *on-phone* debugger that enables the debugging of the application in real time on the top of the running application itself (see Figure 3.8). Its biggest advantage is that this approach runs on the device and does not need any interaction with the developing system. GROPG provides traditional debugging actions like breakpoints, step into, step over, step out, in-scope variable analysis and thread analysis.

[3] `Lint` homepage `http://developer.android.com/tools/help/lint.html`, 2013.
[4] Bell Labs homepage `http://www3.alcatel-lucent.com/wps/portal/belllabs`, 2013.
[5] GROPG homepage `http://cseweb.uta.edu/~tuan/GROPG/`, 2013.

Figure 3.8: GROPG interface.

3.3 Automated Debugging and Testing

Statistical debugging is a type of automated debugging where statistical models of a program's success/failure are used to track program bugs. These models expose the correlation between program behaviour and success or failure of an execution. This type of approaches guides the developers to the root causes of the bugs, by correlating program misbehaviours with the failures and presenting the suspiciousness of a given component being faulty.

In this section will be presented a statistical debugging approach, SFL, as well as some statistical debugging tools.

3.3.1 Spectrum-based Fault Localization (SFL)

Spectrum-based Fault Localization (SFL) [AZGvG09, AZvG07] is a statistics-based lightweight fault localization technique and it is considered to be amongst the most effective ones [AZGvG09, LFY+06, WWQZ08]. This technique uses a dynamic analysis approach, as it relies on program execution information (*program spectrum*) from previous runs (passed and failed) to correlate the software components with the observed failures and determine the likelihood of each component being faulty. Passed runs are executions of a program that completed correctly, while failed runs are executions in which an error was detected. A *Program spectrum* is a collection of data that indicates which components of the software were hit during a run [AZvG07].

The input of the SFL is constituted by the hit spectra and an error-vector (see Figure 3.9). The hit spectra of N runs constitutes a binary $N \times M$ matrix A. M represents the instrumented

components of the program. The error-vector has N-length and represents the information of which runs passed or failed.

$$\begin{array}{c} \\ N \text{ spectra} \end{array} \begin{array}{c} \overset{M \text{ components}}{\overbrace{\phantom{\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1M} \end{bmatrix}}}} \\ \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1M} \\ a_{21} & a_{22} & \cdots & a_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & \cdots & a_{NM} \end{bmatrix} \end{array} \begin{array}{c} \text{error} \\ \text{vector} \\ \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_N \end{bmatrix} \end{array}$$

Figure 3.9: SFL input.

After receiving the input, SFL quantifies the resemblance between each column of the matrix A with the error-vector. The more a column resembles with the error-vector, the greater is the suspiciousness of the corresponding software component being faulty. To do this, SFL relies on *similarity coefficients* like s_J (Jaccard coefficient), s_T (the coefficient used in the Tarantula tool 3.3.2) and s_O (Ochiai coeficient, see Equation 3.1). The latter is considered to be one of the best performing similarity coefficients for fault localization [AZGvG09].

$$s_O(j) = \frac{n_{11}(j)}{\sqrt{(n_{11}(j)+n_{01}(j)) \cdot (n_{11}(j)+n_{10}(j))}} \tag{3.1}$$

where $n_{pq}(m)$ is the number of runs in which a component (m) was hit (p = 1) or not hit (p = 0) during an execution, and where that execution failed (q = 1) or was successful (q = 0). $n_{pq}(m)$ is formally defined as

$$n_{pq}(m) = |\{i \mid a_{ij} = p \wedge e_i = q\}| \tag{3.2}$$

3.3.2 Tarantula

Tarantula[6] [JH05] is a tool used to debug projects written in C, developed at Georgia Tech, that relies on SFL. The visual interface is one of Tarantula's key points (see Figure 3.10), as it highlights each line of code accordingly to their suspiciousness of being faulty (from red for maximum suspiciousness to green for minimum suspiciousness).

3.3.3 EzUnit

EzUnit[7] was developed at University of Hagen and is a statistical debugging tool that aims at the debugging of Java projects, using JUnit test cases. It highlights each line of code accordingly to their suspiciousness of being fault (see Figure 3.11), using colours from red (higher suspiciousness) to green (lower suspiciousness). It also provides a call-graph view that uses the same colour schema (see Figure 3.12).

[6]Tarantula - Fault Localization via Visualization homepage `http://pleuma.cc.gatech.edu/aristotle/Tools/tarantula/`, 2013.

[7]EzUnit – Easing the Debugging of Unit Test Failures homepage `http://www.fernuni-hagen.de/ps/prjs/EzUnit4/`, 2013.

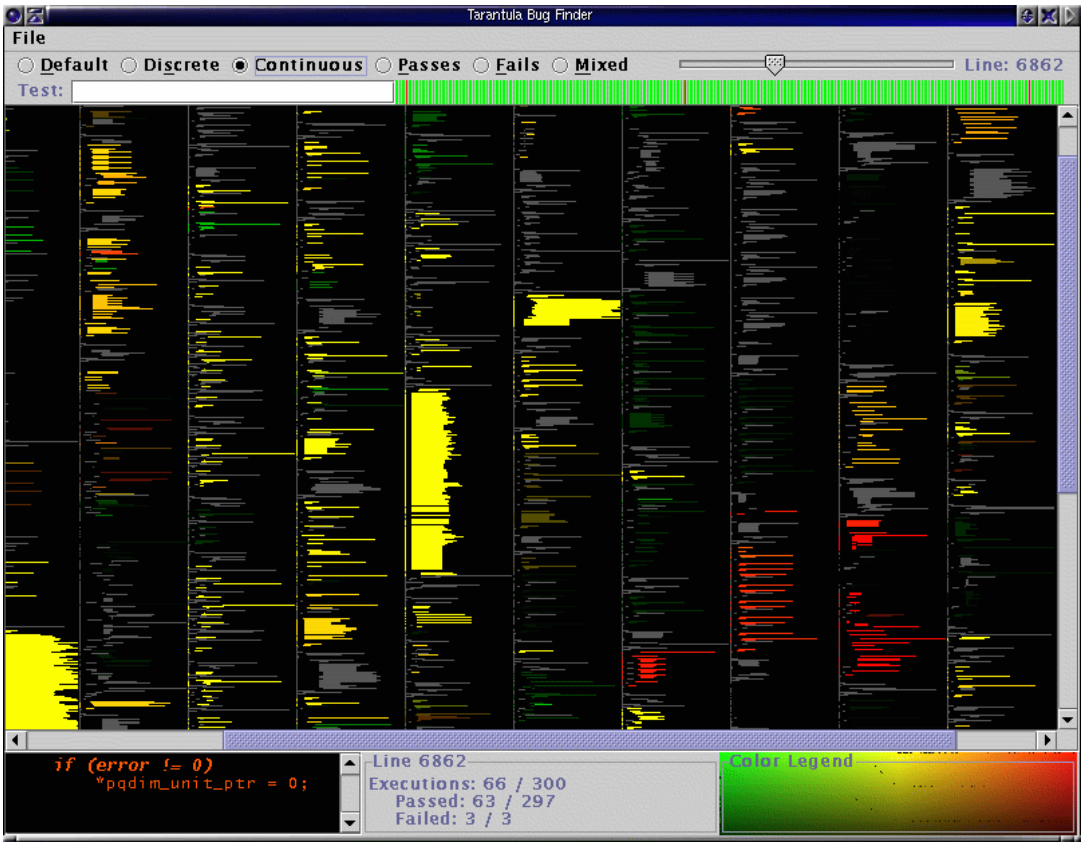


Figure 3.10: Tarantula interface

3.3.4 Zoltar and GZOLTAR

Zoltar is a fault localization tool for C/C++ projects (see Figure 3.13) and it has been developed at Delft University of Technology (TUDelft). It was the base of Rui Abreu's PhD thesis [Abr09] and it won the *Best Demo Award prize at The 24th IEEE/ACM International Conference on Automated Software Engineering (ASE'09) with the publication Zoltar: A Toolset for Automatic Fault Localization* [JAG09].

GZOLTAR[8] [CRPA12] is Java implementation based on Zoltar. It is a framework aimed at debugging Java projects and provides powerful hierarchical and interactive visualizations like sunburst and treemap (see figure 3.14). Like some of the aforesaid tools, GZOLTAR also uses a colour range from red (most probable cause of failure) to green (least probable cause of failure) and implements SFL with the Ochiai coefficient (Equation 3.1).

Besides fault localization, GZOLTAR also provides a test suit reduction and prioritization tool dubbed RZoltar. This tool minimizes the size of the original test suite using constraint-based approaches.

The GZOLTAR tool was the base of André Riboira's MSc thesis [Rib11] and it is the base for the work detailed on this thesis, that aims at adapting this tool to the automatic debugging of Android applications.

[8]The GZOLTAR Project – `http://www.gzoltar.com/`

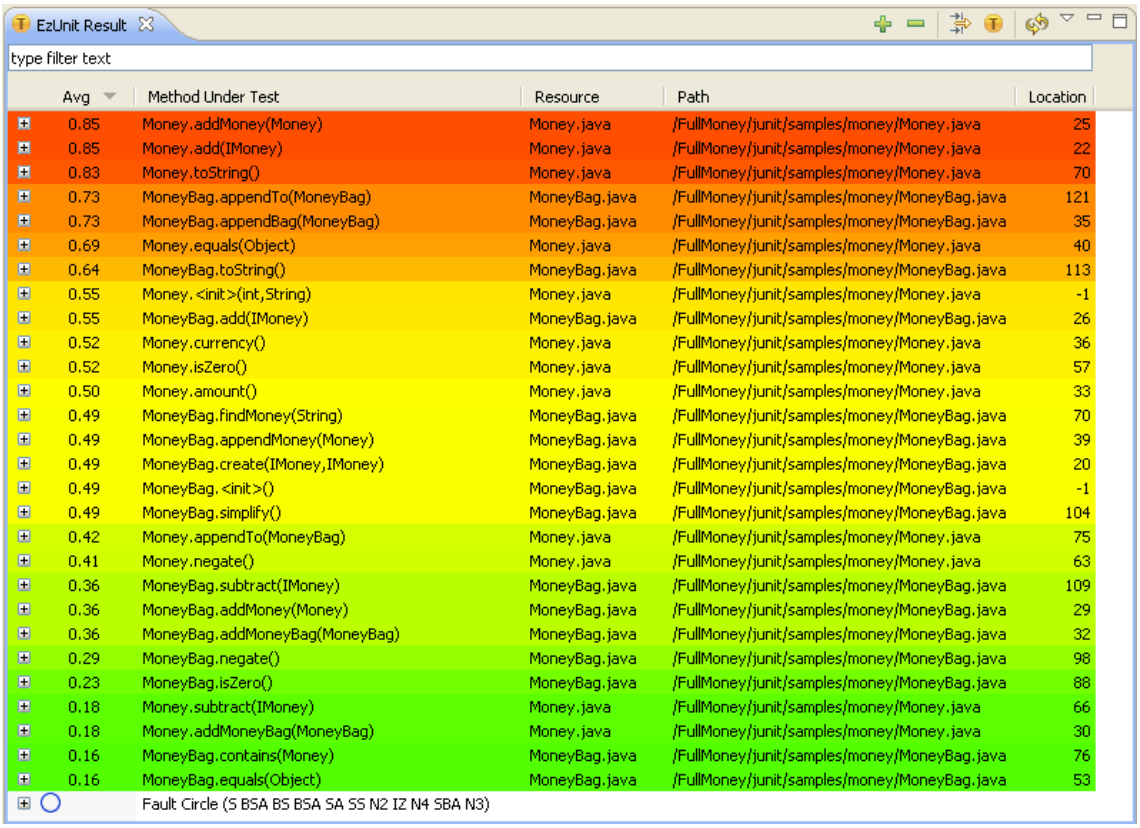


Figure 3.11: EzUnit ranking.

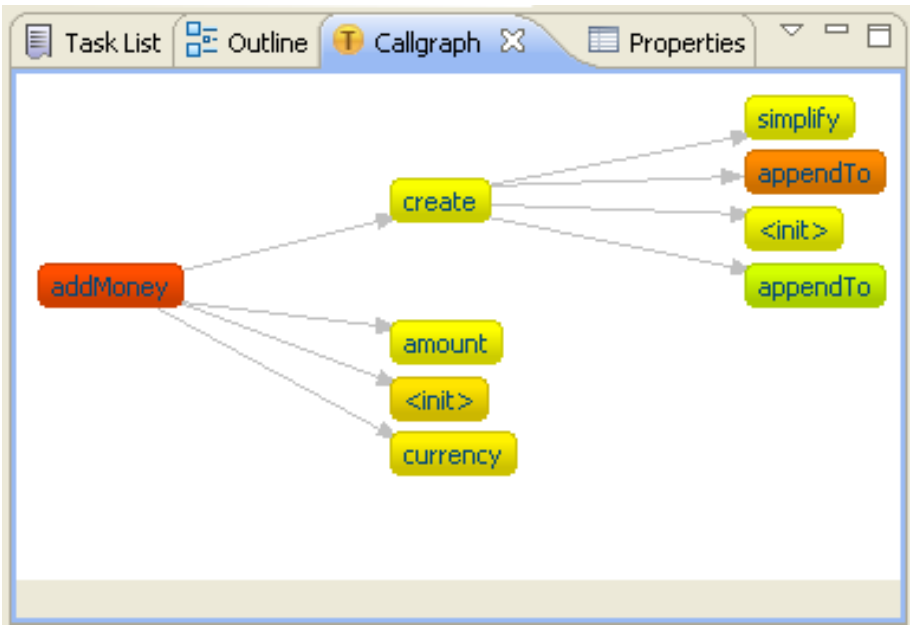


Figure 3.12: EzUnit call graph.

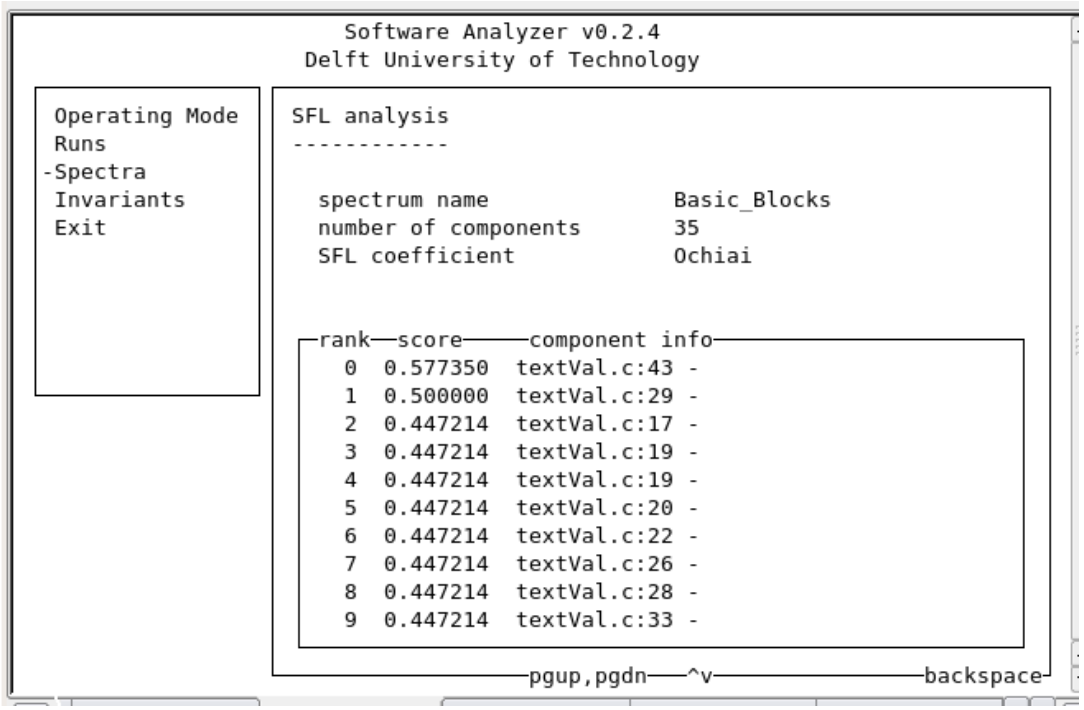


Figure 3.13: Zoltar interface [JAG09].

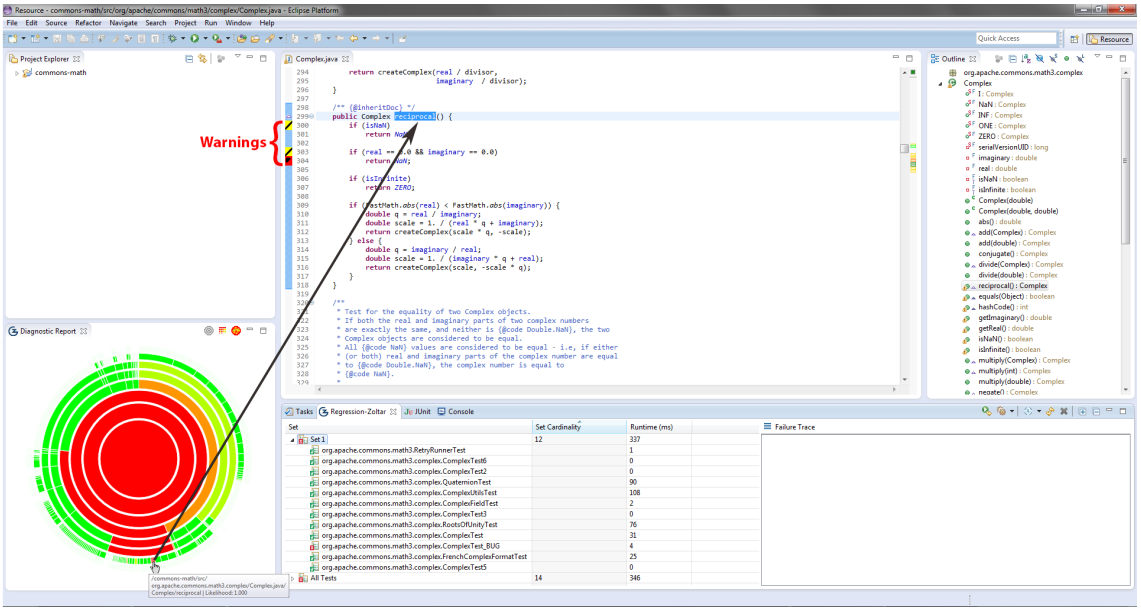


Figure 3.14: GZOLTAR [CRPA12] overview.

3.3.5 Mobile approaches

Attempts to automate the process of testing and debugging mobile apps include the following. Bo Jiang *et al.* described a statistical fault localization technique for mobile embedded systems [JLG+11], where not only the code is targeted, but also suspicious context providers. By incorporating a fault localization logic into the app, so when it crashes, it is capable of choosing the most reliable context provider. The new context provider is chosen from a list of the same group of providers, ordered based on their suspiciousness score.

A system to automatically and systematically generate input events to exercise smartphone apps and its underlying algorithm, based on a concolic testing approach, are described in [ANHY12]. Moreover, GUI testing in mobile devices is an active research subject [HN11, TKH11, AFT+12, JKK+09].

Pascual *et al.* used a generic algorithm to automatically generate optimal application configurations, based on feature model, at runtime [PPF13]. This optimizes the configuration of the system at runtime according to the available resources. The approach does not entail excessive overhead, and helps the app coping with the resource constrained environment and optimizing its performance.

Embedded systems, category in which mobile devices can be fit, were already targeted to measure SFL's performance in such resource-constrained environments [ZAGvG07]. This study has confirmed that fault diagnosis through analysis of program spectra perform well under harsh conditions and opened corridors to new applications, such as run-time recovery.

Chapter 4

Methodologies

To successfully implement the proposed tool, using SFL, several options must be taken into account in each one of its operating phases (see Figure 4.1). In a simplified manner the process is as follows. First the System Under Test (SUT) must be instrumented (Section 4.1) and then the tests must run in order for the tool to be able to retrieve the program spectra (Section 4.2). Finally the tool must analyse the collected data and present the results to the user. Some other analysis can also be included in this process (Section 4.3).

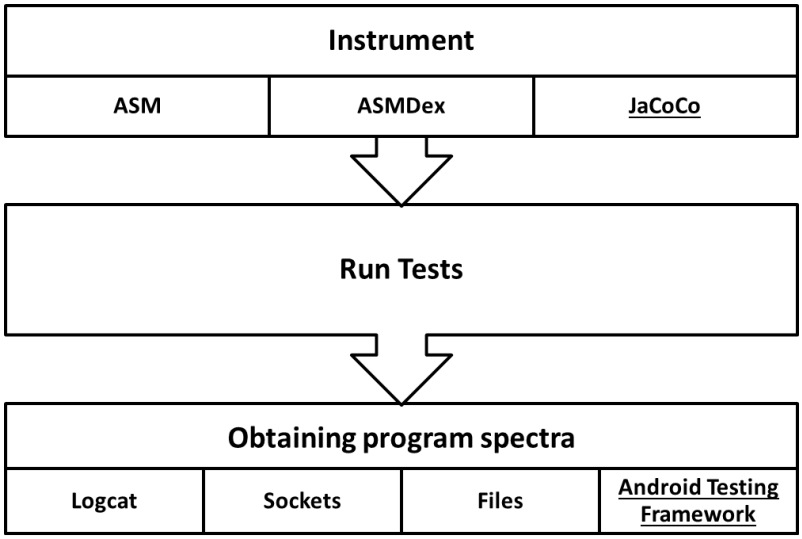


Figure 4.1: Tool execution phases.

In this chapter, the options available for each phase will be presented. Each one of them will be briefly detailed and analysed, trying to find out which one fits best.

4.1 Code Instrumentation

Code instrumentation is a technique used to monitor and analyse runtime information like traces, performance or even code-coverage. It is usually implemented in the form of code instructions

that, for example, output the information or send it to a monitoring tool. There are two types of instrumentation: source instrumentation and binary instrumentation.

Instrumenting is needed to provide the tool with the needed information about the SUT, allowing it to yield useful visual information to the user. Two tools, ASM and ASMDex, both using binary instrumentation, were considered for this work and will be presented in this section.

4.1.1 ASM

ASM[1] is a Java bytecode manipulation and analysis framework [BLC02] and was developed by OW2[2], an open-source software community. It was designed and implemented for performance and be as small and as fast as possible, which makes it a good option for dynamic systems. ASM can be used to modify classes or generate them dynamically.

ASM was successfully used in GZOLTAR's implementation, with the same purposes it was considered in this project. This fact represents an advantage given the knowledge obtained with previous usages.

Advantages:

- Experience from previous usage;
- JVM is stack-based;
- Well documented api.

Disadvantages:

- Lack of usage examples.

4.1.2 ASMDex

ASMDex[3] is also a bytecode manipulation and analysis framework and is also developed by OW2[2]. But, unlike ASM, it handles Dalvik bytecode (used by the DVM as explained in Chapter 2) instead of Java bytecode.

There are a few usage differences between ASM and ASMDex, but both are used essentially the same way. Nevertheless, the virtual machines addressed by both tools have specificities that make all the difference between the mentioned tools. While JVM is stack-based, DVM is register-based. This means that, because ASMDex does not provide any automatic register allocation method, there is a possibility of causing program misbehaviour with the injected instructions during the instrumentation phase. This fact discourages the use of ASMDex, in favour ASM.

Consequently, as detailed in Section 2.4.3, the best approach is to instrument the generated .class files using ASM and let the SDK handle the apk building (and consequent transformation

[1] ASM - Java bytecode manipulation and analysis framework homepage `http://asm.ow2.org/`, 2013.

[2] OW2 Consortium homepage `http://www.ow2.org/`, 2013.

[3] ASMDex - Dalvik bytecode manipulation and analysis framework homepage `http://asm.ow2.org/asmdex-index.html`, 2013.

of class files into .dex files). That way, the problems associated with the register allocations are mitigated.

Advantages:

- Centralized file (all classes are in the same .dex file, compressed into de .apk file);
- Well documented API.

Disadvantages:

- DVM is register-based;
- Lack of usage examples.

4.1.3 JaCoCo Offline Instrumentation

JaCoCo[4] Offline Instrumentation[5] was developed to cope with situations where on-the-fly instrumentation (using an agent) is not suitable, such as:

- Runtime environments that do not support Java agents.
- Deployments where it is not possible to configure JVM options.
- Bytecode needs to be converted for another VM like the Android DVM.
- Conflicts with other agents that do dynamic classfile transformation.

With JaCoCo Offline instrumentation, class files must be pre-instrumented and replace the original files. Moreover, a `jacocoagent.jar` must be present in the application's classpath. This file contains the agent responsible for dumping code coverage information to a file or by TCP. The coverage information files created by the agent can be analysed using the JaCoCo API. This instrumentation can replace the EMMA[6] instrumentation used by the Android Testing framework to perform code coverage analysis[7]. This way, the coverage file is dumped into the device and can easily be retrieved through ADB (see Subsection 3.2.2).

Advantages:

- Coverage information can be dumped to a file or through TCP;
- Replaces the EMMA instrumentation used by the Android Testing framework;
- Is updated on a regular basis;

[4] Acronym for Java Code Coverage.

[5] JaCoCo Offline instrumentation homepage `http://www.eclemma.org/jacoco/trunk/doc/offline.html`, 2013.

[6] EMMA code coverage homepage `http://emma.sourceforge.net/`, 2013.

[7] EMMA has not been updated since 2005, while JaCoCo is updated on a regular basis.

- Provides an easy retrieval of the coverage file;
- Straightforward analysis through the JaCoCo API.

Disadvantages:

- Files must be pre-instrumented;
- `.class` files are instrumented instead of `.dex` files.

4.2 Collection of program spectra

As mentioned in Subsection 3.3.1, a program spectrum is code-coverage information of a given run, used as an input to SFL. Unlike GZOLTAR, where the SUTs are common Java programs, in this project, the nature of the SUTs and the platform itself represent a considerable amount of restraints. To obtain the program spectra, some options have been considered and will be detailed in this section.

4.2.1 LogCat

LogCat is the Android logging system and it can be used to output specific information throughout the code. It has been detailed in section 3.2.1.

Using the LogCat system as a way to transmit program spectra information to the tool was the first considered option. LogCat seemed like the best solution for this problem as it is fast, getting the output information is not too complicated and the instrumentation required in order to use it is fairly simple. However, after implementing this approach and analysing the results, some discrepancies were noticed. These discrepancies were due to the fact that LogCat as a priority-based way of deciding which messages should be shown. So, when any message had an higher degree of priority over the messages originated by the instrumented code, some of the key messages were lost. This would change the results and affect the tool effectiveness. Thus, LogCat approach was dumped and other options have been considered since then.

Advantages:

- good for performance when compared with other approaches;
- simple instrumentation;
- simple to get the information.

Disadvantages:

- priority-based approach which can discard key information.

4.2.2 Sockets

Another one of the weighted approaches was the using of network sockets. The communication between the device and the development system can be made through ADB using an USB cable. A tcp port forward is done so the socket opened in the device can communicate with the server socket opened on the tool and transmit the program spectra information.

Although this is a good approach and it is still at stake, it has a big constraint. For it to be used, it is necessary to inject a new class to the bytecode, where the sockets would be created and the communication with the server would be made. However, synchronization problems are a reality. After Android tests are completed, the system forces the application to shut down, which creates the need to send all the information before that happens. As the network operations cannot be done in the UI thread, they have their own thread, which will be killed along with the application's process. This set or problems creates serious time and synchronization constraints.

Advantages:

- simple to set up communication between the device and the developing system;
- simple to get the information.

Disadvantages:

- fairly complicated instrumentation;
- time constraints;
- needs the network or internet permissions declared on the application's manifest;
- synchronization problems.

4.2.3 Files

The creation of files containing the program spectra was another considered approach. It is very similar to the sockets approach. During each test run, a file would be created containing the spectrum information about a run. After each run, the file would be retrieved through ADB and then analysed. Synchronization would not be a problem, once file operations do not need their own thread in Android. However, new constraints arise. The information contained in the files can, depending on the size of the SUT, occupy a big amount of space, which can be a problem given the known space constraints of mobile devices. On the other hand, the retrieval of the file can represent a performance setback.

Advantages:

- no synchronization problems;
- simple to get the information.

Disadvantages:

- performance issues;
- only works on rooted devices;
- space constraints.

4.2.4 JTAG Boundary Scan Test

JTAG, or Joint Test Action Group, is the common name for the IEEE 1149.1 [IEE90]. This standard defines circuitry that may be built into an integrated circuit to assist in the test, maintenance, and support of assembled printed circuit boards. The circuitry includes a standard interface through which instructions and test data are communicated.

Although JTAG's early applications targeted board level testing, it is becoming broadly used to debug software that runs in embedded systems. As the OCD (On-chip debugger) is accessible through the JTAG interface, it is possible to retrieve trace information about running applications [JCCR12]. For this purpose is necessary to use a JTAG controller as an intermediate to connect to the system under test.

This approach makes it possible to eliminate the instrumentation, thus it can represent a big performance improvement. Android devices are equipped with JTAG ports that are commonly used by the community to fix devices with hardware problems.

Although this may seem to be a good approach to obtain program spectra, the entailed disadvantages rushed its disposal. First of all, although Android devices provide a JTAG port it is internal and located in the motherboard and the only possibility of accessing it is by opening the device. That being said, is not possible to use this approach with emulators. Furthermore, the need to use an extra device (JTAG controller) to retrieve the program spectra, would make the tool difficult to access by the common developer, as the said devices are considerably expensive.

Advantages:

- Android devices have JTAG ports;
- does not require instrumentation, thus eliminating instrumentation overhead.

Disadvantages:

- cannot be used in emulators;
- JTAG port is located on the device's motherboard, so to access it is necessary to open the device;
- requires an expensive JTAG controller.

4.2.5 Android Testing Framework

Android Testing Framework makes use of EMMA to perform code coverage analysis. However, JaCoCo instrumentation (described in Section 4.1.3) can be used instead. Code must be pre-instrumented and, when the test runs ends, the coverage file is dumped into the device. Then, the file is retrieved through ADB (see Subsection 3.2.2) and analysed using the JaCoCo API.

Advantages:

- EMMA instrumentation can be replaced with JaCoCo instrumentation;
- Easy retrieval of the coverage file through ADB;
- Straightforward analysis through the JaCoCo API.

Disadvantages:

- None.

4.3 Other analysis

Besides the main flow of the process, some other analysis can be included to improve the quality of the diagnostic provided.

4.3.1 Lint Analysis

`Lint`, thoroughly described in Section 3.2.7, is a static analysis tool included in Android SDK, which indicates potential bugs in the source of an application. The information provided by lint may be used as an addition to the SFL dynamic analysis, this way including Android specific problems in the diagnostic report.

Advantages:

- Can provide the user additional and useful information;
- Can be customised with new/customized checks.

Disadvantages:

- Static analysis can represent a scalability issue.

4.3.2 Permission Analysis

Android permissions are a security enforcement mechanism, as briefly described at Section 2.4.3. Sometimes, when the developer forgets to declare a given permission, the application behaviour may not be the expected, as an example, if the developer forgets to declare the internet permission

and then the application tries to use that resource, the device will act like if there was no connectivity. The permissions themselves (or their lack) can cause failures or misbehaviours, therefore, permission analysis must also be addressed. To that purpose, the tool should be able to understand what permissions would the application need to behave correctly.

However the enforcement mechanism, where the declaration of required permissions is verified, cannot be accessed by the tool, so there is no way to dynamically know what permissions the application needs. Therefore a static code analysis must be performed and, with the aid of a permission map [FCH+11], it is possible to map the used API methods with the required permissions. That way the tool can provide the user useful information about the required permissions.

Advantages:

- Can provide the user additional and useful information.

Disadvantages:

- Static analysis can represent a scalability issue;
- Poor permission documentation;
- The available map is based on Android 2.2.

Chapter 5

Tooling

There are a few toolsets offering spectrum-based fault localization, such as GZOLTAR [CRPA12], Tarantula [JH05], or EzUnit [BKMS07]. GZOLTAR and Tarantula show the diagnostic reports visually in an attempt to facilitate the quest for the defects. EzUnit provides a textual ranking of the lines that are most likely to be faulty and assigns a background color to each line matching its failure score.

In this section, the novel aspects of MZOLTAR, which make it suitable to the mobile apps testing and debugging phase, are detailed. Since MZOLTAR is based on GZOLTAR, MZOLTAR provides the same set of visualizations as GZOLTAR, but enhanced with the distinctive aspects of the Android apps development. Figure 5.1, as an example, shows one of the three GZOLTAR's visualizations[1], the Sunburst visualization. Sunburst shows the information as an hierarchical structure, taking advantage of the fact that software is inherently hierarchical, in particular the Java-based object-oriented software used in the development of Android applications. Both in GZOLTAR and MZOLTAR, the visualizations also allow users to navigate through the code and interact with the source code, trying to ease the task of finding bugs.

5.1 Motivational Example

To illustrate the problem addressed in this paper, consider the simple Android application in Figure 5.2 (based on the example used in [GSPGvG10]). To improve the legibility, the coverage matrix and the error detection vector were transposed.

This running example uses a function *count()* that receives a string as an argument and prints the number of times each type of char (letter, number or other) occurs in that string. A bug has been injected in line 3 (see Figure 5.2), where the *let* counter should be incremented by just one when the string includes a capital letter, but instead it is being incremented by two. The figure also shows the code coverage information of the 8 executed tests. For each row, a ● appears in the columns that correspond to the tests where that line was executed. The error vector, *e* is presented at the bottom of the table, showing the passed/failed information of the executed tests. Resorting

[1] Other visualization can be seen at `http://www.gzoltar.com`

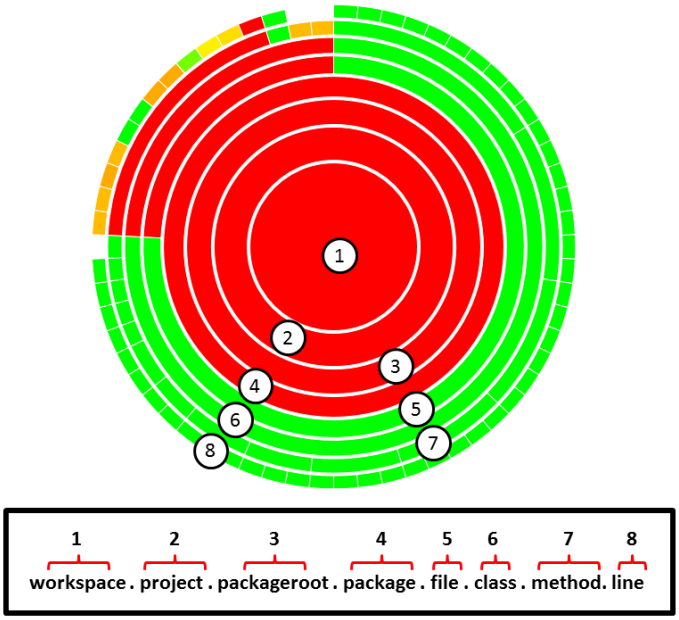


Figure 5.1: Sunburst visualization (and how to interpret it).

to this information, the Ochiai coefficient, s_O, is used to calculate the suspiciousness of a given line containing a fault. In this case, SFL has successfully performed the fault localization as the ranking created encourages the developer to inspect the faulty line first.

In Figure 5.3 the graphical user interface of the implemented Android application is presented. The application receives a string through a text box and shows the result of the counting algorithm. Also, the result is affected by the bug injected in the previous example. There are 4 letters in the string, but the *letter* counter indicates the value 5, as the capital letter 'A' increments 2 units in the *letter* counter while it should increment just one. The application was also tested using the Android testing framework[2] to assess its functioning.

As explained in Chapter 2, Android applications are not only composed by the source files containing the underlying application logic, but also include layout files, translation files and a manifest file that presents the application information to the Android system. Moreover, the embedded nature of Android devices hinders the retrieval of runtime information, used as input to SFL.

In this application, there are two source files: *CharacterCount.java* and *CharacterCountActivity.java*. *CharacterCount.java* has the Java implementation of the mentioned example, and *CharacterCountActivity.java* is an Android Activity responsible for the interaction with the user, that is, receiving the string and presenting the result of each counter. For the Android system to be able to start CharacterCountActivity, it must be declared in the applications' manifest. If the activity is not declared, the application will not be able to start, and the tests cases fail without reporting any code coverage information. This scenario, a common mistake in Android development, is also

[2]Android Testing Fundamentals `http://developer.android.com/tools/testing/testing_android.html`, 2013.

Subject: CharCount		T										
		t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	s_O
	`class CharCount {...`											
	`static void count(String s) {`											
1:	`int let, dig, other;`	●	●	●	●	●	●	●	●	●	●	0.63
2:	`for(int i = 0; i < s.length(); i++) {`	●	●	●	●	●	●	●	●	●	●	0.63
3:	`char c = s.charAt(i);`	●	●	●	●	●	●	●		●	●	0.67
4:	`if ('A'<=c && 'Z'>=c)`	●	●	●	●	●	●	●		●	●	0.67
5:	`let += 2; /* FAULT */`	●		●	●			●				1.00
6:	`else if ('a'<=c && 'z'>=c)`	●	●	●	●	●	●	●		●	●	0.67
7:	`let += 1;`					●	●	●		●	●	0.22
8:	`else if ('0'<=c && '9'>=c)`	●	●	●	●	●	●			●	●	0.53
9:	`dig += 1;`	●	●	●	●		●			●	●	0.57
10:	`else if (isprint(c))`					●	●					0.00
11:	`other += 1; }`					●	●					0.00
12:	`System.out.println(let + " " + dig + " " + other); }`	●	●	●	●	●	●	●	●	●	●	0.63
	`...}`											
Test case outcome (pass=✓, fail=✗)		✗	✓	✗	✗	✓	✓	✗	✓	✓	✓	

Figure 5.2: Example of SFL technique with Ochiai coefficient (adapted from [GSPGvG10]).

shown in Listing 5.1 where the activity element is commented. This way, it is also important to, somehow, assess these problems directly related to the Android system functioning. Not declaring an Activity in the manifest file is only an example, among many other issues related with Android particularities that can cause failures in the execution of an application. These particularities pose interesting challenges, such as:

- Given it is a resource-constrained environment, is SFL, and collecting the coverage information, well suited to perform automatic fault localization in mobile apps software?
- SFL only considers components of the software that has been executed, and therefore it is important to reason with the non-executable files (such as the *manifest* and resource files). Hence, can we include non-executable components in the analysis?

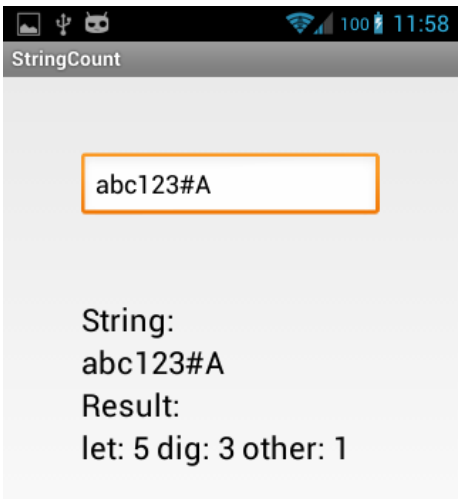


Figure 5.3: Android example application with bug in result.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="example.charactercount"
    android:versionCode="1"
    android:versionName="1.0" >
    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="17" />
    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <!-- Without this activity the program will fail
        <activity
            android:name="example.charactercount.CharacterCountActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity> -->
    </application>
</manifest>
```

Listing 5.1: Android Manifest sample code.

5.2 Combining Static and Dynamic Analysis

Given Android idiosyncrasies and to be able to find as many defects as possible, it is important to combine static analysis (report produced by `Lint`) with the dynamic analysis (report produced by SFL). As outlined in Subsection 3.3.1, the SFL technique used in the context of this thesis, computes a suspiciousness value between 0 and 1 for each component. As the outcome of these two analysis is to be combined into a single suspiciousness value per component, we devised a function, so-called *`Lint` coefficient*, that maps the priority and severity reported by `Lint` into a value between 0 and 1.

To support the definition of the `Lint` coefficient, let $I(m)$ be the sequence of all issues found for component m, $I_S(m)$ be a sequence of severities for component m, and $I_P(m)$ be a sequence of priorities for component m. $I_S^{(i)}(m)$ and $I_P^{(i)}(m)$ refer to the i^{th} issue. Severities are mapped into a numerical value according to: *Warning* maps to 1.0, *Error* to 1.1, and *Fatal* to 1.2. The rationale is that *Errors* contribute more (we considered 10% more) than *Warning* for the suspisciousness of the component being faulty (i.e., the weight factor should be higher), and Fatal contribute an extra 10%. Note, however, that this mapping can be parameterized by the developers. Using these

priorities and severities, the weighted arithmetic mean of the number of issues found per severity is calculated. Formally,

$$\bar{I}_P(m) = \frac{1}{W_T(m)} \cdot \sum_{i=0}^{|I(m)|} I_P^{(i)}(m) \cdot I_S^{(i)}(m) \tag{5.1}$$

where $W_T(m)$ is the sum of all severities for component m, and is defined as

$$W_T(m) = \sum_{i=0}^{|I(m)|} I_S^{(i)}(m)$$

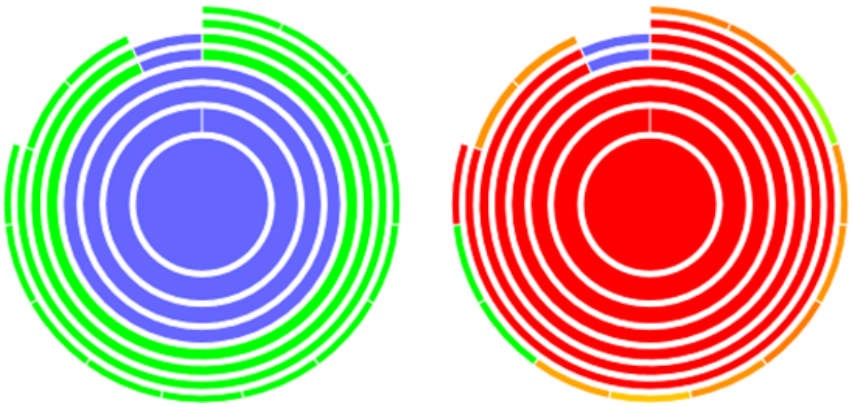
and $|I_S(m)|$ is the number of issues reported by `Lint` for component m. The `Lint` coefficient is the normalized (to assume values between 0 and 1) weighted arithmetic mean measure of the priorities per severity, amplified by the severity. Normalization is achieved by dividing $\bar{I}_P(m)$ with maximum value for priority ($maxPr = 10$, as explained in Subsection 3.2.7). Suppose there are two components, for which the weighted arithmetic mean of the priorities is 0.6, one of them containing only warnings, and the other one containing only fatals. As they do not represent the same threat, the yielded priority value must be amplified by the corresponding severity to reflect the real threat. Formally,

$$L_c(m) = min\left(\frac{\bar{I}_P(m)}{maxPr} \cdot \bar{I}_S(m),\ 1\right) \tag{5.2}$$

where $\bar{I}_S(m)$ is the arithmetic mean of all severities for component m, defined as

$$\bar{I}_S(m) = \frac{W_T(m)}{|I(m)|}$$

As there are two coefficients quantifying the suspiciousness of a given component being faulty, the challenge is now to successfully make them co-exist within the same visualization without hindering its intuitiveness and added value. We have decided to use a new color scheme (viz. blue) to representing the static analysis outcome. Figure 5.4 illustrates how the combination works in practice.



(a) Static defects only. (b) Static and dynamic defects.

Figure 5.4: `Lint` integration in visualizations.

Finally, the visualization still needs to account for the fact that the `Lint` and SFL coefficients may both be different than 0 (i.e., the component has both static-related concerns and dynamically it was also observed, to some extent, correlation with the failures). As mentioned by the Information Foraging theory [Pir07, LBB+13, FSP+13], the theory we relied upon when building the debugging visualizations, to maximize the information gain per interaction's cost, it is necessary to provide the user with the cues that best indicate the existence of a defect. To give the user a better perception of the component's state, we used a combination of the two coefficients to generate the component's color in the visualization. Instead of just using the Ochiai coefficient $s_O(m)$, the suspiciousness for the component is *amplified* with the `Lint` Coefficient as follows

$$C(m) = min\Big(s_O(m) \cdot (1 + L_c(m)),\ 1\Big) \tag{5.3}$$

5.3 Workflow

As mentioned in Section 5.1, Android devices present a series of challenges in what it concerns the application of SFL. To surpass those challenges, MZOLTAR relies on the `Android testing framework` and `JaCoCo`[3] to run the applications' tests and acquire coverage information. As MZOLTAR is a plugin for the Eclipse Integrated Development Environment (IDE), it also relies on the abstractions provided by the ADT plugin to perform some of the tasks. MZOLTAR's flow is the following:

1. Instrument bytecode;
2. Generate application's `apk` and flush into the device;
3. Run tests;
4. Collect code coverage;
5. Run diagnostic report.

To instrument the application, phase (1), there were three options available, as stated in Chapter 4: instrument the Dalvik Bytecode directly in the `apk` file or instrument the original Java Bytecode generated by Eclipse (afterwards compiled into the referred Dalvik Bytecode when the `apk` is built). As Dalvik VM is register based and the available frameworks (such as ASMDex[4]) do not provide a way to automatically reallocate the registers after instrumenting the code, we opted for the Java Bytecode instrumentation. In particular, we used a feature of `JaCoCo`, the Offline instrumentation (see Subsection 4.1.3) that also relies on the `ASM` framework. Android test framework has an EMMA code coverage[5] analysis feature that can be used in conjunction with `JaCoCo` offline instrumentation to retrieve the code coverage information of a test execution.

[3] JaCoCo homepage `http://www.eclemma.org/jacoco/`, 2013.
[4] ASMDex homepage `http://asm.ow2.org/asmdex-index.html`, 2013.
[5] EMMA homepage `http://emma.sourceforge.net/`, 2013.

After the Bytecode was instrumented, ADT API was used to build the `apk` file, in phase (2), as well as to run the tests in phase (3). In this last step the code coverage information is generated in the device. In phase (4), the IDevice interface provided by ADT is used to pull the coverage data file from the device to be further processed, offline, by MZOLTAR. Using the `JaCoCo` API, the coverage information is mapped into the input expected by SFL and the diagnostic report is computed (phase 5).

5.4 Eclipse integration

MZOLTAR is offered as a plugin for the well-known Eclipse IDE. As the visualizations are integrated in the environment and MZOLTAR takes advantage of some of the features provided by the Eclipse IDE (such as code navigation and editor markers), the debugging effort is reduced. Also, the provided cues try to be as explicit and well-aimed as possible, to help the user debug accurately. The existence of an official ADT plugin to Eclipse was taken into consideration, as it aided in the implementation of some of the features (see Subsection 5.3). There follows a description of MZOLTAR's main features.

Visualizations such Sunburst (see Figure 5.1) provides useful information to the user, as they translate the diagnostic reports into a much more intuitive graphical representation. The similarity coefficient value of each component is used to create a color gradient, that goes from red, for the components that are more likely to be faulty, and ends with green to the less likely ones. The structured visual representation also makes it easier to understand the program structure, hence reducing the effort of locating a given component in the code. The available visualizations and their features are thoroughly described in [GCA13].

Besides the intuitive representation, some interactions were also implemented to ease the debugging process. to get to a component's location in the code, the user only needs to click on that component in the visualization. Then, the editor opens the file and highlights the previously clicked component. It is also possible to change the visualization, by performing a root change or zooming, so the user can focus on a desired set of components.

Before being able to execute the diagnostic algorithm, the user has to (i) select the project that is going to be tested, (ii) select the Test Runner to use, and (iii) select whether or not the application should be removed from the device after it is tested (see Figure 5.5). Furthermore, we decided to use the ADT device chooser dialog to make the experience of using MZOLTAR as similar as possible with the usual experience of developing an Android application with the Eclipse IDE (see Figure 5.6).

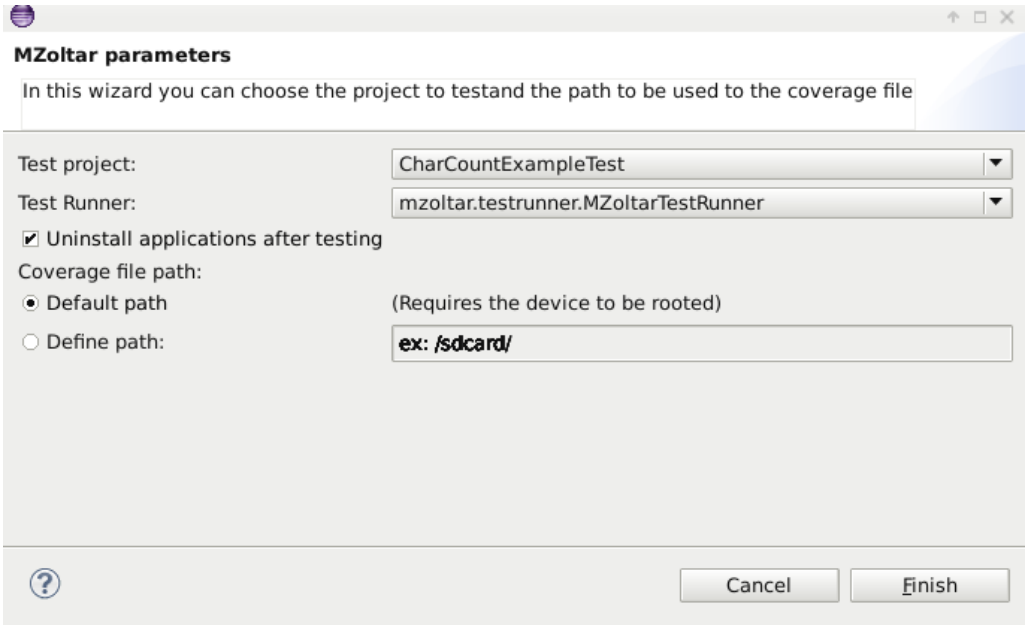


Figure 5.5: MZOLTAR parameters' interface.

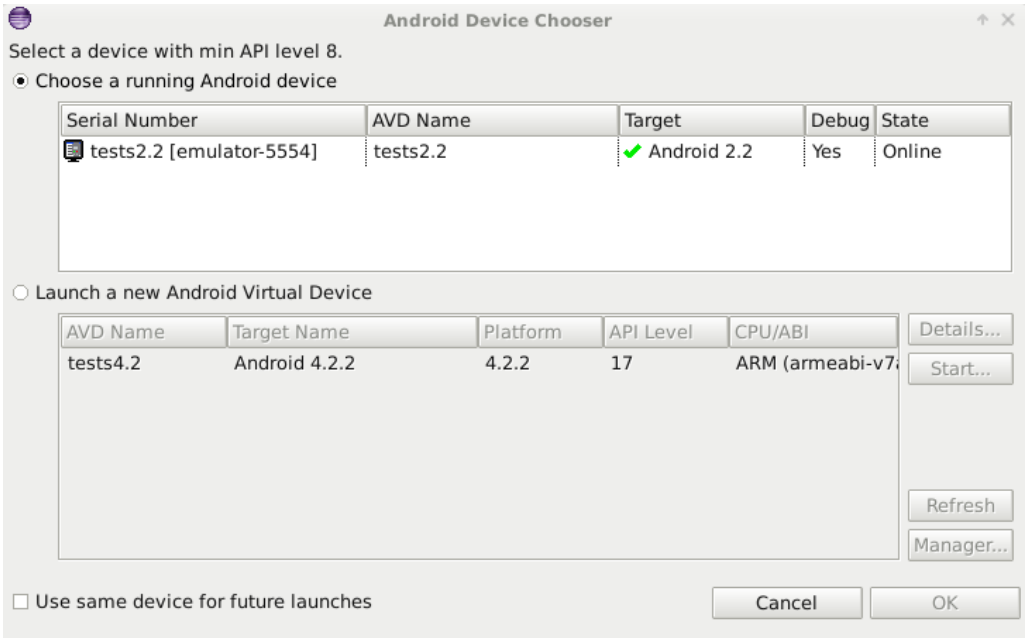


Figure 5.6: MZOLTAR device chooser.

Chapter 6

Empirical Evaluation

In this section, the empirical evaluation carried out to assess MZOLTAR's performance is described, in particular to verify its applicability to the context of mobile apps. We start by describing the experimental setup, followed by a discussion on the observed results. The empirical evaluation aims at answering the following research questions:

RQ1 Is the MZOLTAR's instrumentation overhead negligible?

RQ2 Does MZOLTAR yield accurate diagnostic reports under Android device's constrained environment?

RQ3 Does the integration with `Lint` contribute to a better diagnostic quality?

6.1 Experimental Setup

Table 6.1: Experimental Subjects.

Subject	Version	Line Of Code (LOC)	Test Cases	Test Cases LOC	Coverage	Resources LOC
CharCount	1.0	148	10	133	92.2%	115
ConnectBot	1.7.1	32911	14	484	0.7%	7673
Google Authenticator	2.21	3659	170	2825	76.6%	5275
StarDroid	1.6.5	13783	187	3029	29.7%	2694

Four mobile apps, of different sizes and complexities, were considered to empirically evaluate MZOLTAR. Table 6.1 presents further information about the following subjects:

- CharCount is the subject used as a motivational example in Section 5.1;
- ConnectBot[1] is an Android Secure Shell (SSH) client;

[1] ConnectBot homepage `http://code.google.com/p/connectbot`, 2013.

- Google Authenticator[2] is a two-step authentication application;
- StarDroid[3] is sky map open source project.

To foster reproducibility and comparability, we report the version number of the subjects used in the evaluation. Line Of Code (LOC) count information was obtained using `Code Analyzer`[4]. The tests provided with the applications were used to conduct the experiments and no new tests were implemented. Android testing framework was used to run the tests and code coverage information was obtained using `JaCoCo`[5], enabling the `coverage` flag of the Android tests framework. Then, we used the `EclEmma`[6] Eclipse plugin to analyse the generated coverage files.

As the subjects are bug-free (with regard to the test suites), eight common mistakes [HGP09] were injected in each subject. To facilitate the activation of the faults, thus automating the testing process, we built a fault injection framework. This framework allows to enable/disable the faults automatically and use a custom InstrumentationTestRunner[7], named MZoltarTestRunner, that parses an argument (`injectedFaults`) which indicates which faults should be active per run. Then, each application was executed 30 times for each of the following scenarios $\{(f,g) \mid f \in \{1,2,3,5\} \wedge g \in \{1,2,3,5,10\}\}$, where f is the number of injected faults and g is the number of tests considered in a transaction. A transaction is composed by all the tests that are executed when the VM is initiated until its tear down. The reason for this notion of transaction (and not one per test case) is that to obtain the individual code coverage information for each test, only one test can be executed per run (technological limitation). Instead of considering each SFL matrix line as just one test, each line represents all the tests in a transaction. Concerning the pass/fail information, a transaction fails when one of its tests fail and passes when all its tests pass. Hence, creating a new VM per test case may impose a considerably high overhead, as a delay is observed between each test execution since the system reboots the VM where the tests are run. This delay was measured to be approximately one second per execution.

These scenarios make it possible to assess the performance of MZOLTAR in different situations, being also important to evaluate the tradeoff *effectiveness* vs *time*. Running each test separately may entail a considerable time overhead (since, per test execution, Android terminates and starts a new VM). Our goal is to evaluate the consequences of executing several tests simultaneously, assessing the potential time reduction vs. the potential information and effectiveness losses.

The experiments target device was an emulator running Android 2.2 (API Level 8) with a 4" (480x800 hdpi) screen, an ARM processor, 343MB of RAM and 32MB of VM Heap. The emulator was used on a 3.16GHz Intel® Core™ 2 Duo PC with 2GB of RAM, running Debian 7.0 (wheezy).

[2]Google Authenticator homepage `http://code.google.com/p/google-authenticator`, 2013.
[3]StarDroid homepage `http://code.google.com/p/stardroid`, 2013.
[4]Code Analyzer homepage `http://www.codeanalyzer.teel.ws`, 2013.
[5]JaCoCo homepage `http://www.eclemma.org/jacoco`, 2013.
[6]EclEmma homepage `http://www.eclemma.org/`, 2013.
[7]Android Instrumentation Test Runner homepage `http://developer.android.com/reference/android/test/InstrumentationTestRunner.html`, 2013.

6.2 Evaluation Metric

To measure the success of a diagnosis technique we use the diagnostic quality C_d, which estimates the number of components the tester needs to inspect to find the fault [SFA13]. Note that C_d cannot be computed prior to computing the ranking: one does not know the actual position of true-fault candidates in the ranking beforehand. Because multiple explanations can be assigned with the same similarity value s_O, the value of C_d for the real fault d_* is the average of the ranks that have the same similarity value:

$$\theta = |\{j | s_O(m) > s_O(d_*)\}|,\ 1 \leq j \leq M$$

$$\phi = |\{j | s_O(m) \geq s_O(d_*)\}|,\ 1 \leq j \leq M \tag{6.1}$$

$$C_d = \frac{\theta + \phi - 1}{2}$$

In the multiple fault cases we use the one-at-a-time mode, discussed in [SFA13]: one fault is identified and fixed, and then the fault localization process is repeated (including a re-run of the test suite and a re-computation of the input for the fault localization technique). We report C_d for the first fault found, as one can estimate the impact of reducing the number of faults on C_d in the experiments of lower number of injected faults.

We further use a metric $\bar{\rho}$, the *density of the coverage matrix* [GSPGvG10], that has been used in the past to build confidence on the C_d obtained, and understand whether one can still improve the diagnostic report by adding more tests. The density of a coverage matrix is the average percentage of components covered by test cases. It is defined as follows

$$\bar{\rho} = \frac{\sum_{i=1}^{N} \sum_{j=1}^{M} a_{ij}}{N \cdot M}$$

where N and M denote the number of test cases and the number of components, respectively. a_{ij} represents the coverage of the component m when the test t_i is executed. Values of $\bar{\rho}$ close to 0 means that test suite touch a small parts of the program, whereas values close to 1 means that test suite tend to cover most components of the program. In [GSPGvG10] it has been shown that $\bar{\rho} = 0.5$ is the best for fault localization, provided that there is a *diversity* in the test cases of the suite.

6.3 Experimental Results

In this section the experimental results and their relation to the research questions are further discussed.

RQ1: Is the MZOLTAR's instrumentation overhead negligible?

Table 6.2 shows the execution times for the test subjects. The average execution times of the mobile apps' instrumented versions are, as expected, slightly higher then the original versions. The collected results show that the used instrumentation entails an average time overhead of 5.75% (with standard deviation σ=2.49). We cannot claim that we have not altered the timing behaviour of the system, but the applications, although slightly slower, still function properly. Therefore, we conclude that the instrumentation overhead is not prohibitive.

Table 6.2: Execution times.

Subject	Original	Instrumented	Overhead
CharCount	1.82s	1.86s	2%
ConnectBot	1.25s	1.35s	8%
Google Authenticator	80.49s	87.26s	8%
StarDroid	14.70s	15.46s	5%

RQ2: Does MZOLTAR yield accurate diagnostic reports under Android device's constrained environment?

Figure 6.1 plots the diagnostic accuracy C_d for the following number of injected faults: 1, 2, 3, and 5. The injected faults are 8 faults that are considered to be common [HGP09]. For the single fault scenario, the reported results are on average for the 8 faults, whereas for the multiple fault scenarios we have randomly repeated the experiments 30 times (randomly injecting the faults). For all the test subjects the average diagnostic accuracy obtained is constant and maintains a low value, even for the multiple fault scenarios. One has to inspect an average of 5 components before finding a bug. This facts show that SFL performs well in terms of diagnostic accuracy despite the resource constraints imposed by the Android architecture. We can also conclude that the existence of multiple faults does not affect the diagnostic accuracy.

As mentioned before, SFL takes as input the coverage of a transaction (concept described in Section 6.1). To address this potential bottleneck, we considered to execute multiple test cases per virtual machine. On the one hand, there is a potential reduction in the runtime overhead, but, on the other hand, the noise implied by considered multiple tests as one execution may entail information loss. Consequently, worsening the diagnostic quality C_d. By grouping test cases, i.e., increasing the number of test cases per transaction, the number rows in spectra matrix, N, is decreased, consequently increasing the matrix density, ρ, which is an explanation for the degradation of the diagnostic quality. Figures 6.2 and 6.3a plots the impact of grouping test cases in ρ and C_d. For $\rho \geq 0.5$, there is a significant worsening of the diagnostic quality (cf. [GSPGvG10]).

Figure 6.2 plots C_d when grouping, randomly, several tests per execution (namely, 1,2,3,5, and 10), and the execution overhead is plotted in Figure 6.3b. For CharCount and ConnectBot,

the C_d increases with the number of tests executed per transaction, while for GoogleAuth and StarDroid C_d remains practically constant. These differences are explained by the number of tests each application provides. This way, the percentage reduction of the number of lines (caused by clustering of several tests in each run) of the spectra matrix is higher in the subjects with less tests implemented, thus worsening C_d. Regarding the execution overhead, an exponential reduction was observed with the increase of the number of tests. This overhead reduction is explained by the fact that there is no need to restart the VM.

As an example, when executing 10 test cases per transaction, we observed that grouping test cases reduced the execution overhead in 79% on average (σ=8.36), at the cost of a loss in the diagnostic quality of 74% (σ=12.14). This is mainly due to the density growth that comes along with the increase of the number of tests in a group.

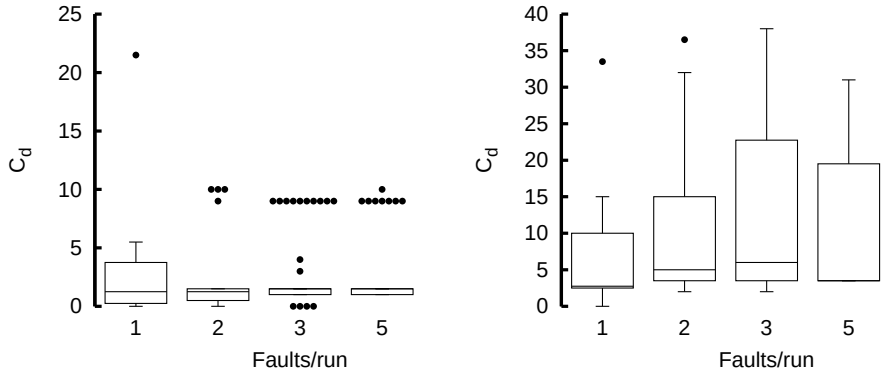
RQ3: Does the integration with `Lint` contribute to a better diagnostic quality?

As said before, there are Android-specific bugs that cannot be explained by the SFL's dynamic analysis. We have injected two static related defects into each subject to investigate the added value of adding the static analysis yielded by `Lint`. The injected faults are: lack of activity registry in the manifest, incorrect cast of a view, or using methods not available in the API version in usage.

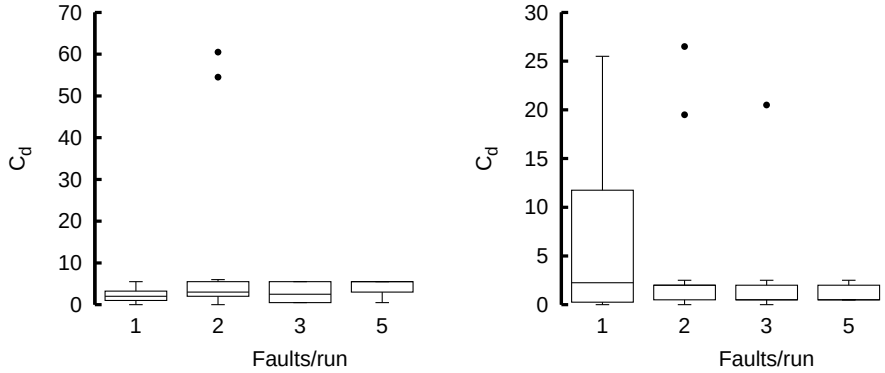
Table 6.3 shows that none of the injected defects were pinpointed by SFL, therefore entailing a high C_d (as in the worst case the entire program would have to be inspected before deciding to look into the Android specific problems). Compared against SFL only, combining both static and dynamic analysis led to an averaged reduction of 99.9% ($\sigma = 0.21$).

Table 6.3: C_d comparison with and without `Lint`.

Subject	**SFL**		**SFL + `Lint`**	
	Bug 1	**Bug 2**	**Bug 1**	**Bug 2**
CharCount	148	148	0	1
ConnectBot	32911	32911	30	19
Google Authenticator	3659	3659	3	1
StarDroid	13783	13783	18	12

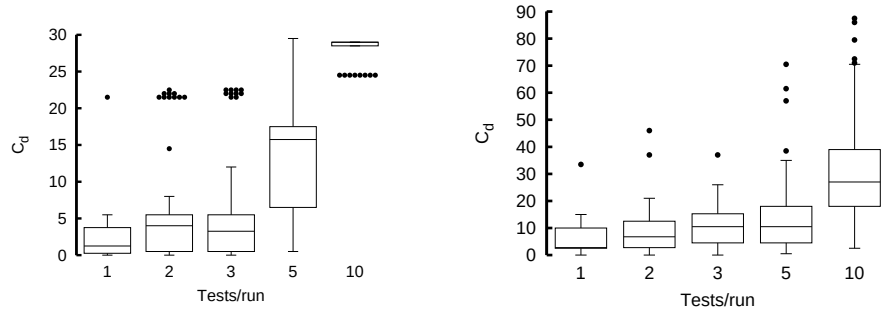


(a) CharCount (b) ConnectBot

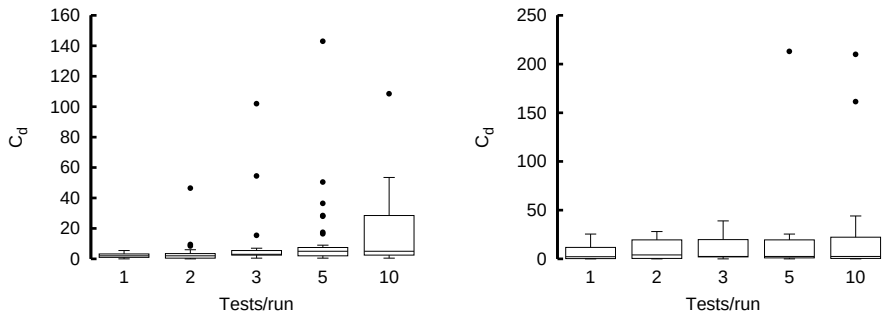


(c) Google Authenticator (d) StarDroid

Figure 6.1: Diagnostic accuracy C_d.

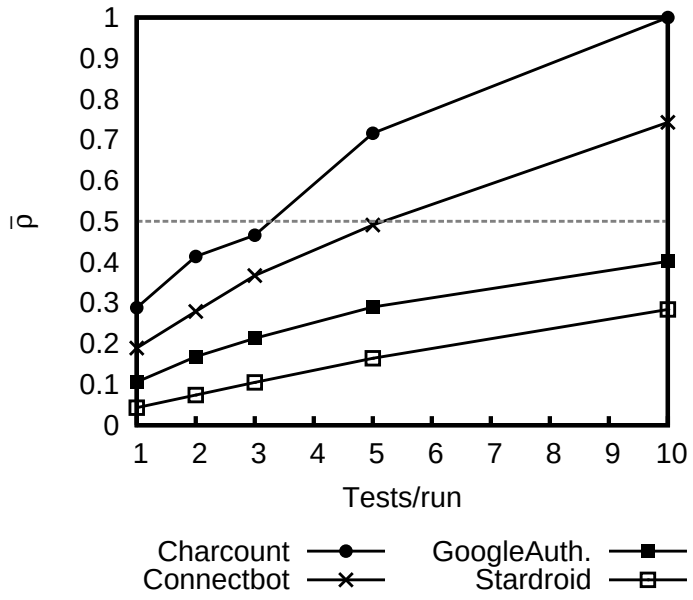


(a) CharCount (b) ConnectBot

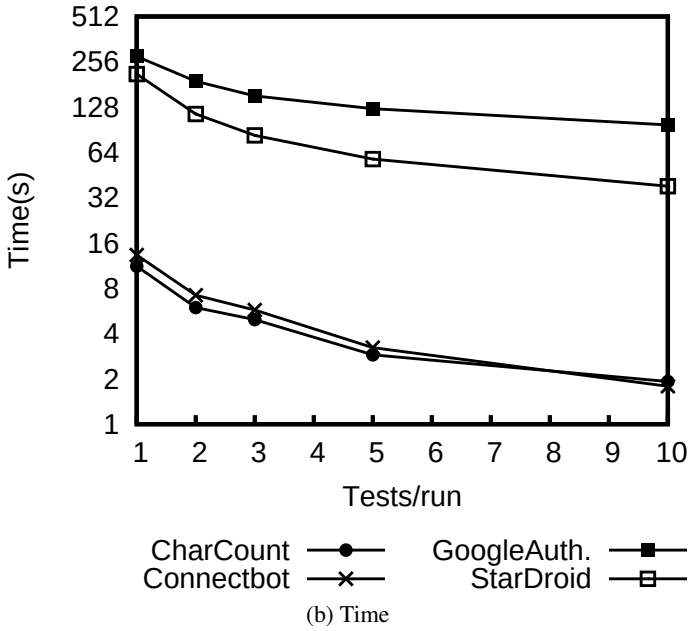


(c) Google Authenticator (d) StarDroid

Figure 6.2: Impact of Grouping Test Cases on C_d.



(a) Density



(b) Time

Figure 6.3: Outcomes when grouping multiple tests per transaction.

Chapter 7

Conclusions

7.1 Related Work

After analysing briefly the state of the art of Debugging Tools, it is possible to conclude that the tools available to debug Android applications imply a manual, time consuming and somehow complex task. Although they are nowadays used in the development of Android applications, it is obvious that an automatic debugging approach would ease the debugging process and increase the quality and reliability of such applications.

Despite the myriad of techniques and approaches, there are still shortcomings when applying these techniques in the context of mobile, resource-constrained apps. Available automated fault localization toolsets do not offer easy integration into the mobile apps world. As a consequence, and although Graphical User Interface (GUI) testing automation for mobile systems is an active research topic, manual approaches are still prevalent in the mobile apps debugging and testing phases, and the debugging tools available for mobile apps only offer manual debugging features [EMK13]. Hence, MZOLTAR addresses that issue by providing an automated fault localization approach, offering the dynamic analysis provided by SFL combined with the information yielded by `Lint` static analysis.

7.2 Methodologies

To adapt SFL to Android applications, a set of options have been analysed and compared, to understand which ones fit best in each phase of the process. There's no perfect solution in any of the options analysed, thus these analysis help minimize the implementation risks. The analysis also helped in the elimination of some of the options considered like ASMDex in the instrumentation phase as well as LogCat and JTAG at the spectrum acquisition phase.

Variables like scalability, efficiency and effectiveness must be the top priorities. So, performance setbacks, instrumentation overheads and information loss are some of the points that should have a big weight when deciding which approaches should be used.

Taking this criteria into account and the fact that it can be used with the Android Testing Framework, `JaCoCo` offline instrumentation was selected in the instrumentation phase. Therefore, we used the Android Testing Framework feature that allows the generation of coverage information files, then retrieving it through ADB and analysing the coverage information using the `JaCoCo` API. Using `JaCoCo` and the Android Testing Framework presented as the best and strongest solution, as it relied on the well documented and easily usable `JaCoCo` API and on the official Testing Framework of our target System.

Finally, `Lint` static analysis was included to improve the diagnostic provided, by addressing the more particular issues presented by Android. The permission analysis was not used because its permission basis is outdated and could not present the same results throughout the different versions of Android.

7.3 Lessons Learned

During the implementation of the toolset described in this thesis, some interesting facts were uncovered and are worth mentioning. Most of the following points are related with technological limitations of the Android system:

- Although ASMDex was the first and expected choice to instrument Android applications, it was the first to be ruled out. This happened because the DVM is register based and ASMDex does not provide any way to automatically re-allocate the registers. This way, it is not a trivial task to instrument all the lines without changing the correct functioning of the application.

- Before building the .dex file included in the application package, Eclipse compiles the Java code into common .class files, which enables the usage of several bytecode instrumentation frameworks.

- It is possible to communicate between an Android device and its host machine through USB. The communication is made through TCP sockets and the server must be located on the device. The reason behind this is that ADB forwards the requests from the host computer to the device and not the other way around, thus it is not possible to initiate the communication from the device. Once established, the communication is bi-directional.

- The code coverage analysis performed by the Android Testing framework can be used with JaCoCo instrumentation instead of EMMA instrumentation. `JaCoCo` is updated with bug fixes and new features on a regular basis, while EMMA's last stable version goes back to 2005.

- ADB commands have a limit of 1024 characters. If the command is too large, an error will occur and the command will not be run.

7.4 Empirical Evaluation Results

An empirical study using 4 real open-source Android applications (with single and multiple injected faults), confirms that spectrum-based fault localization is well suited for the mobile apps development context. The infrastructure to collect the information needed is lightweight (overhead of $5.75\% \pm 2.49\%$ on average), while the diagnostic accuracy is similar to the one observed on general-purpose applications [Abr09]. It was shown that the developer only has to inspect an average of 5 components before finding the bug. Additionally, we investigated the possibility of grouping test cases to further reduce the execution time, achieving an average reduction of $79\% \pm 8.36\%$), at the cost of a loss in the diagnostic quality of $74\% \pm 12\%$.

Also, the added value of integrating `Lint`'s static analysis into MZoltar was demonstrated, as there are Android-specific defects which SFL cannot blame. Concerning these Android-specific defects, the concerned approach was able to reduce the number of components the tester needs to inspect to find the fault by $99.9\% \pm 0.21\%$ on average.

7.5 Main Contributions

The main contributions of this thesis are:

- The discussion of the challenges faced by developers when doing fault localization, highlighting the real-world relevance of the problem;
- A fully automated approach for localizing defects in Android applications. Our approach is based on a well-known spectrum-based fault localization technique, plus with `Lint` static analyser provided with Android SDK, and produces a visual report to aid in locating the defects;
- A coefficient (`Lint` coefficient L_c) , mapping the report yielded by `Lint` to a $0-1$ scale, to (i) quantify the suspiciousness of a component being faulty and (ii) be able to integrate it the the dynamic fault localization results;
- A toolset, dubbed MZOLTAR,embedded into the Eclipse IDE providing the proposed fault localization technique. The toolset will be available at `http://gzoltar.com/mzoltar`;
- An empirical study to demonstrate the efficiency of MZOLTAR and and verify its applicability to the context of mobile apps.

To the best of our knowledge, the combination of using static and dynamic analysis, as done in MZOLTAR, in the context of mobile apps has not been described before.

7.6 Publications

With the work done in this thesis, we were able to submit to the following conferences:

- Pedro Machado, José Campos and Rui Abreu. **Automatic Debugging of Android Applications** – Accepted for publication at *The 6th Meeting of Young Researchers of University of Porto (IJUP'13)*, 2013. This paper outlines the idea of developing an automated fault localization tool that provides valuable diagnosis to the developer when debugging mobile applications using SFL.

- Pedro Machado, José Campos and Rui Abreu. **MZoltar: Automatic Debugging of Android Applications** – Accepted for publication at *First International Workshop on Software Development Lifecycle for Mobile*, 2013. This paper describes the MZOLTAR tool and its underlying architecture.

- Pedro Machado, José Campos and Rui Abreu. **Combining Static and Dynamic Analysis in Mobile Apps Fault Localization** – Submitted to *The 24th IEEE International Symposium on Software Reliability Engineering*, 2013. This paper describes both the MZOLTAR tool, as well as the integration of SFL dynamic analysis integration with information from static analysis provided by `Lint`.

7.7 Future work

Future work includes the following. First, we intend to provide MZOLTAR in the recently announced Android Studio IDE. Finally, we plan to port MZOLTAR to other mobile technologies, notably to iOS and Windows Phone. However, while the original research questions have been successfully addressed and solved, some other questions have emerged and are worthy of reference. Although the results achieved in the context of this thesis are promising, there is still a lot to understand and improve.

`Lint` yields the issues, found in a given project, that may potentially result in bugs. However, not all of the flagged issues may be considered in the context of runtime failures, as they will never cause the application to fail. Moreover, some issues may only cause failures when very specific criteria are met, thus reducing their potential to result in bugs. Hereupon, understanding which issues may cause runtime failures and what should be their real weight in the diagnosis is one of the points we intend to address. Additionally, it would be interesting to understand what are the most common mistakes in the development of Android applications, as reported in [HGP09] for general-purpose applications. To the best of our knowledge, such work has not been reported yet.

Furthermore, the colour scheme added to the visual reports, described in Section 5.2 may still have room to improve. Although the diagnostic results were good, the empirical evaluation did not cover the visualizations. Hence, the best way to assess the usefulness of the new colour scheme is to conduct a user study and ascertain, from a set of different colour schemes and patterns, which one would be the most beneficial to the developers.

After, we intend to investigate the addition of a bug prediction approach to MZOLTAR, such as the one described in [KZWJZ07]. By flagging spots that recurrently require bug fixes, bug prediction tries to guess whether a piece of code is potentially buggy or not. Google reported

an interesting approach[1] where additionally to bug prediction scores, the weight of a bug-fixing commit is reduced based on how old it is.

[1] Bug prediction at Google http://google-engtools.blogspot.co.at/2011/12/bug-prediction-at-google.html, 2013.

References

[Abr09] Rui Abreu. *Spectrum-based Fault Localization in Embedded Software*. PhD thesis, Delft University of Technology, November 2009.

[AFT+12] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. Using GUI ripping for automated testing of Android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE '12, pages 258–261, New York, NY, USA, 2012. ACM.

[ALRL04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, January 2004.

[ANHY12] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok 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.

[AZGvG09] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *J. Syst. Softw.*, 82(11):1780–1792, November 2009.

[AZvG07] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. On the Accuracy of Spectrum-based Fault Localization. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, TAICPART-MUTATION '07, pages 89–98, Washington, DC, USA, 2007. IEEE Computer Society.

[BKMS07] Philipp Bouillon, Jens Krinke, Nils Meyer, and Friedrich Steimann. EZUNIT: a framework for associating failed unit tests with potential programming errors. In *Proceedings of the 8th international conference on Agile processes in software engineering and extreme programming*, XP'07, pages 101–104, Berlin, Heidelberg, 2007. Springer-Verlag.

[BLC02] E. Bruneton, R. Lenglet, and T. Coupaye. Asm: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, 30, 2002.

[CCSH12] Outsourcing Capgemini: Consulting, Technology, Sogeti, and Hewlett-Packard (HP). World Quality Report 2012. Technical report, Capgemini, September 2012.

[CRPA12] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. GZoltar: An Eclipse Plug-in for Testing and Debugging. In *Proceedings of the 27th IEEE/ACM*

International Conference on Automated Software Engineering, ASE 2012, pages 378–381, New York, NY, USA, 2012. ACM.

[dK09] J. de Kleer. Diagnosing multiple persistent and intermittent faults. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI'09)*, pages 733–738, 2009.

[EMK13] Mona Erfani, Ali Mesbah, and Philippe Kruchten. Real Challenges in Mobile App Development. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, ESEM '13, New York, NY, USA, 2013. ACM.

[FCH+11] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.

[FSP+13] Scott D. Fleming, Christopher Scaffidi, David Piorkowski, Margaret M. Burnett, Rachel K. E. Bellamy, Joseph Lawrance, and Irwin Kwan. An Information Foraging Theory Perspective on Tools for Debugging, Refactoring, and Reuse Tasks. *ACM Transactions on Software Engineering and Methodology*, 22(2):14, 2013.

[GCA13] Carlos Gouveia, José Campos, and Rui Abreu. Using HTML5 Visualizations in Software Fault Localization. In *Proceedings of the 1st IEEE Working Conference on Software Visualization*, VISSOFT '13, Washington, DC, USA, 2013. IEEE Computer Society.

[GSPGvG10] Alberto Gonzalez-Sanchez, Eric Piel, Hans-Gerhard Gross, and Arjan J. C. van Gemund. Prioritizing Tests for Software Fault Localization. In *Proceedings of the 2010 10th International Conference on Quality Software*, QSIC '10, pages 42–51, Washington, DC, USA, 2010. IEEE Computer Society.

[HGP09] M. Hamill and K. Goseva-Popstojanova. Common trends in software fault and failure data. *Software Engineering, IEEE Transactions on*, 35(4):484–496, 2009.

[HN11] Cuixiong Hu and Iulian 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.

[HS02] B Hailpern and P Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12, 2002.

[HZZ+09] Dan Hao, Lingming Zhang, Lu Zhang, Jiasu Sun, and Hong Mei. VIDA: Visual interactive debugging. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 583–586, Washington, DC, USA, 2009. IEEE Computer Society.

[IEE90] IEEE. IEEE Standard Test Access Port and Boundary-Scan Architecture. *IEEE Std. 1149.1-1990*, 1990.

[JAG09] T. Janssen, R. Abreu, and A.J.C. Gemund. Zoltar: A toolset for automatic fault localization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 662–664. IEEE Computer Society, 2009.

REFERENCES

[JCCR12] Ricardo Barbosa Joao Carlos Cunha and Gilberto Rodrigues. On the use of boundary scan for code coverage of critical embedded software. In *Proceedings of the 23th International Symposium on Software Reliability Engineering*, ISSRE 2012, 2012.

[JH05] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 273–282, New York, NY, USA, 2005. ACM.

[JKK+09] A. Jaaskelainen, M. Katara, A. Kervinen, M. Maunumaa, T. Paakkonen, T. Takala, and H. Virtanen. Automatic GUI test generation for smartphone applications - an evaluation. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 112–122, 2009.

[JLG+11] Bo Jiang, Xiang Long, Xiaopeng Gao, Zhifang Liu, and W.K. Chan. FLOMA: Statistical fault localization for mobile embedded system. In *Advanced Computer Control (ICACC), 2011 3rd International Conference on*, pages 396–400, 2011.

[KZWJZ07] Sunghun Kim, Thomas Zimmermann, E James Whitehead Jr, and Andreas Zeller. Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering*, pages 489–498. IEEE Computer Society, 2007.

[LBB+13] Joseph Lawrance, Christopher Bogart, Margaret M. Burnett, Rachel K. E. Bellamy, Kyle Rector, and Scott D. Fleming. How Programmers Debug, Revisited: An Information Foraging Theory Perspective. *IEEE Transactions on Software Engineering*, 39(2):197–215, 2013.

[LFY+06] Chao Liu, Long Fei, Xifeng Yan, Jiawei Han, and Samuel P. Midkiff. Statistical Debugging: A Hypothesis Testing-Based Approach. *IEEE Trans. Softw. Eng.*, 32(10):831–848, October 2006.

[Mil11] D.T. Milano. *Android Application Testing Guide*. Community experience distilled. PACKT PUB, 2011.

[MS03] W. Mayer and M. Stumptner. Model-based debugging using multiple abstract models. In *Proceedings of International Workshop on Automated and Analysis-Driven Debugging (AADEBUG'03)*, pages 55–70, 2003.

[MS08] W. Mayer and M. Stumptner. Evaluating Models for Model-Based Debugging. In *Proceedings of International Conference on Automated Software Engineering (ASE'08)*, pages 128–137, 2008.

[NCT13] Tuan A. Nguyen, Christoph Csallner, and Nikolai Tillmann. GROPG: A graphical on-phone debugger. In *Proceedings 35th ACM/IEEE International Conference on Software Engineering (ICSE), New Ideas and Emerging Results (NIER) track*, May 2013.

[Pir07] P.L.T. Pirolli. *Information Foraging Theory: Adaptive Interaction with Information*. Human Technology Interaction Series. Oxford University Press, USA, 1 edition, 2007.

REFERENCES

[PPF13] Gustavo G. Pascual, Mónica Pinto, and Lidia Fuentes. Run-time adaptation of mobile applications using genetic algorithms. In *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '13, pages 73–82, Piscataway, NJ, USA, 2013. IEEE Press.

[Rib11] A. Riboira. *GZoltar: A Graphical Debugging Interface*. MSc Thesis, University of Porto, 2011.

[SF12] F. Steimann and M. Frenkel. Improving coverage-based localization of multiple faults using algorithms from integer linear programming. In *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*, pages 121–130, 2012.

[SFA13] Friedrich Steimann, Marcus Frenkel, and Rui Abreu. Threats to the Validity and Value of Empirical Assessments of the Accuracy of Coverage-Based Fault Locators. In *Procedings of the International Symposium in Software Testing and Analysis*, ISSTA 2013, New York, NY, USA, 2013. ACM.

[SSPC13] Gang Shu, Boya Sun, Andy Podgurski, and Feng Cao. MFL: Method-Level Fault Localization with Causal Inference. In *Proceedings of the IEEE Sixth International Conference on Software Testing, Verification and Validation*, ICST '13, 2013.

[Tas02] G Tassey. The Economic Impacts of Inadequate Infrastructure for Software Testing, 2002.

[TKH11] T. Takala, M. Katara, and J. Harty. Experiences of system-level model-based gui testing of an android application. In *Proceedings of the IEEE Fourth International Conference on Software Testing, Verification and Validation*, ICST '11, pages 377–386, 2011.

[WDLG12] W.E. Wong, V. Debroy, Yihao Li, and Ruizhi Gao. Software fault localization using dstar (d*). In *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on*, pages 21–30, 2012.

[WWQZ08] Eric Wong, Tingting Wei, Yu Qi, and Lei Zhao. A Crosstab-based Statistical Method for Effective Fault Localization. In Rob Hierons and Aditya Mathur, editors, *Proceedings of the 1st International Conference on Software Testing, Verification, and Validation (ICST'08)*, pages 42–51, Lillehammer, Norway, 2008. IEEE Computer Society.

[ZAGvG07] Peter Zoeteweij, Rui Abreu, Rob Golsteijn, and Arjan J. C. van Gemund. Diagnosis of Embedded Software Using Program Spectra. In *Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, ECBS '07, pages 213–220, Washington, DC, USA, 2007. IEEE Computer Society.

[ZPA+08] Peter Zoeteweij, Jurryt Pietersma, Rui Abreu, Alexander Feldman, and Arjan JC Van Gemund. Automated fault diagnosis in embedded systems. In *Secure System Integration and Reliability Improvement, 2008. SSIRI'08. Second International Conference on*, pages 103–110. IEEE, 2008.