

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

# Android Crawler

Marco António Fernandes Gonçalves



**FEUP** FACULDADE DE ENGENHARIA  
UNIVERSIDADE DO PORTO

Mestrado em Engenharia de Software

Supervisor: Ana Cristina Ramada Paiva

July 18, 2019



# **Android Crawler**

**Marco António Fernandes Gonçalves**

Mestrado em Engenharia de Software

Approved in oral examination by the committee:

Chair: Prof. Nuno Honório Rodrigues Flores

External Examiner: Prof. José Francisco Creissac Freitas de Campos

Supervisor: Prof. Ana Cristina Ramada Paiva

July 18, 2019



# Abstract

The current situation is marked by the competition that exists between the several smartphone industries in search of market dominance. Consequently, the market for mobile applications is flooded daily with thousands of new applications, especially the market of mobile applications for the Android platform. The smartphone and its applications have become a tool almost indispensable to the human being in his daily life. The main factor that distinguishes Google Play from other mobile applications stores is the simplicity to add new applications, that is, for an application to stand out from the rest it is necessary to guarantee the quality and proper functioning.

One of the solutions most commonly used in this industry to guarantee the quality and proper functioning of the applications is running tests. A solution to reduce the costs associated with testing is automation, but unfortunately, much work is still needed in this area to create a completely satisfactory approach.

This dissertation involves improving an existing tool, the *iMPAcT Tool*. The approach presented in this dissertation extends the *iMPAcT Tool* with reverse engineering capabilities. The *REiMPAcT* approach is able to extract information about the activities and states traversed by a dynamic exploration of an Android mobile application.

The main objective of this dissertation is to improve some behaviors that are tested by the *iMPAcT Tool* and use the *REiMPAcT* approach to explore the activities of the Android applications, so that in the end it is possible to build a hierarchical finite state machine (HFMSM) with three distinct levels of abstraction.

To conclude, the *REiMPAcT* approach allows users to comprehend better the Android application under analysis and help in software testing context by enabling users to find out the source of the failure within the code. The *iMPAcT Tool* help the industry of Android applications to develop better products.

**Keywords:** Android Crawler, Android Testing, Software Testing, Test Automation



# Resumo

A atualidade é marcada pela competição que existe entre as diversas indústrias de smartphones à procura do domínio dos mercados. Consequentemente, o mercado de aplicações móveis é inundado diariamente com milhares de novas aplicações, especialmente o mercado de aplicações móveis para a plataforma Android. O smartphone e as suas aplicações tornaram-se numa ferramenta quase indispensável ao ser humano para o seu dia-a-dia. O principal fator que distingue a Google Play das restantes lojas de aplicações móveis é a facilidade para adicionar novas aplicações, ou seja, para que uma aplicação se destaque das restantes é necessário garantir a qualidade e o bom funcionamento da mesma.

Uma das soluções mais utilizadas neste setor para garantir a qualidade e o bom funcionamento das aplicações é a execução de testes. Uma solução para reduzir os custos associados aos testes é a automação, mas infelizmente, ainda é necessário muito trabalho nessa área para criar uma abordagem totalmente satisfatória.

Esta dissertação envolve a melhoria de uma ferramenta já existente, a *iMPAcT Tool*. A abordagem apresentada nesta dissertação estende a *iMPAcT Tool* com recursos de engenharia reversa. A abordagem da *REiMPAcT* é capaz de extrair informações sobre as atividades e estados percorridos por uma exploração dinâmica de uma aplicação Android.

O principal objetivo desta dissertação é melhorar alguns comportamentos que são testados pela *iMPAcT Tool* e usar a abordagem *REiMPAcT* para explorar as atividades das aplicações Android, para que no final seja possível construir uma máquina de estados finita hierárquica (HFSSM) com três níveis distintos de abstração.

Em soma, a abordagem *REiMPAcT* permite aos utilizadores compreender melhor as aplicações Android em análise e ajuda no contexto de teste de software, permitindo ao utilizador descobrir a origem da falha dentro do código. A *iMPAcT Tool* ajuda a indústria de aplicações Android a desenvolver produtos melhores.

**Keywords:** Android Crawler, Android Testing, Software Testing, Test Automation





# Acknowledgements

And so my academic journey comes to an end, there are so many people to thank and remember for making part of this journey and always believe it was possible. The truth is that this would not be possible without you guys.

First of all, I would like to thank my parents, my sister, and my grandmother, for all unconditional love throughout my life. Thank you for always supporting me, for trusting me, and for making me the man I am today.

Because life is also made of friends, I would like to thank all my friends for always leading me to the right port, for always hearing my complaints, and for making this last journey a lot more fun!

Last but not least, I would like to thank my supervisor, Prof. Ana Paiva for all the dedication, advice, commitment, and sharing of knowledge that made the elaboration of this dissertation possible.

Marco António Fernandes Gonçalves



*“You can build your own things that other people can use.  
Once you learn that, you’ll never be the same again.”*

Steve Jobs



# Contents

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Introduction</b>                                      | <b>1</b>  |
| 1.1      | Problem . . . . .  | 1         |
| 1.2      | Motivation and Goals . . . . .                           | 2         |
| 1.3      | Methodology . . . . .                                    | 3         |
| 1.4      | Structure of the Dissertation . . . . .                  | 4         |
| <b>2</b> | <b>State of the Art</b>                                  | <b>5</b>  |
| 2.1      | The world of Mobile Applications . . . . .               | 5         |
| 2.2      | Current approaches to test Mobile applications . . . . . | 6         |
| 2.2.1    | Official Frameworks . . . . .                            | 8         |
| 2.2.2    | Non-Official Frameworks . . . . .                        | 9         |
| 2.3      | Model Based Testing for Mobile Application . . . . .     | 10        |
| 2.4      | Pattern Based Testing for Mobile Applications . . . . .  | 10        |
| 2.4.1    | Patterns . . . . .                                       | 10        |
| 2.4.2    | Pattern Based GUI Testing . . . . .                      | 12        |
| 2.5      | Software Reverse Engineering . . . . .                   | 13        |
| 2.5.1    | Reverse Engineering approaches over the years . . . . .  | 13        |
| 2.5.2    | Mobile Reverse Engineering . . . . .                     | 14        |
| 2.6      | Modeling Hierarchical GUIs . . . . .                     | 14        |
| 2.7      | Android Best Practices . . . . .                         | 15        |
| 2.8      | Conclusion . . . . .                                     | 16        |
| <b>3</b> | <b>iMPAcT Tool</b>                                       | <b>17</b> |
| 3.1      | Approach . . . . .                                       | 17        |
| 3.2      | Implementation . . . . .                                 | 18        |
| 3.2.1    | Exploration . . . . .                                    | 18        |
| 3.2.2    | Patterns matching . . . . .                              | 19        |
| 3.2.3    | Testing . . . . .  | 19        |
| 3.2.4    | <i>iMPAcT Tool</i> configuration . . . . .               | 20        |
| 3.2.5    | Types of exploration of the <i>iMPAcT Tool</i> . . . . . | 20        |
| 3.3      | Patterns Catalog . . . . .                               | 21        |
| 3.3.1    | Side Drawer Pattern . . . . .                            | 22        |
| 3.3.2    | Orientation Pattern . . . . .                            | 23        |
| 3.3.3    | Resource Dependency Pattern . . . . .                    | 25        |
| 3.3.4    | Tabs Pattern . . . . .                                   | 25        |
| 3.3.5    | Back Pattern . . . . .                                   | 27        |
| 3.3.6    | Background Pattern . . . . .                             | 27        |
| 3.3.7    | Up Pattern . . . . .                                     | 28        |

|          |  |           |
|----------|--|-----------|
| 3.3.8    | Action bar Pattern . . . . .   | 29        |
| 3.3.9    | Calls Pattern . . . . .  | 30        |
| 3.4      | Visualization of the results . . . . .   | 30        |
| 3.4.1    | Report . . . . .   | 31        |
| 3.4.2    | Finite state machine . . . . .   | 32        |
| 3.5      | Conclusion . . . . .   | 33        |
| <b>4</b> | <b>Implementation</b>  | <b>35</b> |
| 4.1      | Patterns Catalogue . . . . .   | 35        |
| 4.1.1    | Call Pattern . . . . .   | 35        |
| 4.2      | <i>REiMPAcT</i> . . . . .  | 36        |
| 4.2.1    | <i>REiMPAcT</i> architecture . . . . .   | 37        |
| 4.2.2    | <i>REiMPAcT</i> output . . . . .   | 39        |
| 4.3      | Conclusions . . . . .  | 42        |
| <b>5</b> | <b>Validation</b>  | <b>43</b> |
| 5.1      | Research Questions . . . . .   | 43        |
| 5.2      | Technical Specifications . . . . .   | 44        |
| 5.3      | Test Methodology . . . . .   | 45        |
| 5.4      | Finite State Machine . . . . .   | 46        |
| 5.5      | Extracting Activities Dynamically . . . . .  | 47        |
| 5.6      | Percentage of Activities Explored . . . . .  | 48        |
| 5.7      | Exploration Time and Percentage of Activities Explored . . . . .   | 50        |
| 5.8      | Conclusions . . . . .  | 51        |
| <b>6</b> | <b>Conclusions and Future Work</b>   | <b>53</b> |
| 6.1      | Discussion . . . . .   | 53        |
| 6.1.1    | <b>RQ 1:</b> Is the <i>iMPAcT Tool</i> able to build an FSM of an Android application when testing the Call Pattern? . . . . . | 53        |
| 6.1.2    | <b>RQ 2:</b> Is it possible to extract the name of the activities of an Android application by a dynamic process? . . . . .    | 53        |
| 6.1.3    | <b>RQ 3:</b> Which is the percentage of activities explored dynamically within a limited time period? . . . . .                | 54        |
| 6.1.4    | <b>RQ 4:</b> What is the additional percentage of activities explored when exploration time increases? . . . . .               | 54        |
| 6.2      | Future Work . . . . .  | 54        |
| 6.3      | Conclusions . . . . .  | 54        |
| <b>A</b> | <b>Results of the Experiments</b>  | <b>63</b> |

# List of Figures

|      |  |    |
|------|--|----|
| 2.1  | Smartphone user penetration as a percentage of the total global population from 2014 to 2021 [Staf]. . . . .       | 6  |
| 2.2  | The number of available applications in the Google Play Store from December 2009 to September 2018 [Stac]. . . . . | 7  |
| 2.3  | Focus areas for testing mobile applications from 2013 to 2017 [Staa]. . . . .                                      | 7  |
| 3.1  | Finite State Machine of application org.ametro. . . . .  | 18 |
| 3.2  | <i>iMPAcT Tool</i> approach [MPF14]. . . . .   | 19 |
| 3.3  | The interface of the <i>iMPAcT Tool</i> . . . . .  | 21 |
| 3.4  | Example of a <i>Side Drawer</i> pattern [Dra]. . . . .   | 22 |
| 3.5  | Example of possible orientations. . . . .  | 23 |
| 3.6  | Example of a <i>tab</i> [Tab]. . . . .   | 25 |
| 3.7  | Example of a <i>back</i> button. . . . .   | 27 |
| 3.8  | Example of several applications running in <i>background</i> . . . . .   | 28 |
| 3.9  | Example of the <i>Up</i> button. . . . .   | 29 |
| 3.10 | Example of an <i>action bar</i> . . . . .  | 30 |
| 3.11 | Example of incoming call state while using an Android application. . . . .   | 31 |
| 4.1  | The architecture of the reverse engineering approach ( <i>REiMPAcT</i> ). . . . .                                  | 37 |
| 4.2  | ADB Shell command output. . . . .  | 38 |
| 4.3  | Output of the Activities Finder component from the <i>REiMPAcT</i> approach. . . . .                               | 38 |
| 4.4  | Output of the Time Synchronizer component from the <i>REiMPAcT</i> approach. . . . .                               | 39 |
| 4.5  | Hierarchical Finite State Machine (HFSM) with 2nd level detailed for Activity A. . . . .                           | 40 |
| 5.1  | Call Test Pattern output (FSM) for the application Wikipedia. . . . .  | 47 |
| 5.2  | The first level of the HFSM for the application MyResults. . . . .   | 48 |
| 5.3  | The second level of the HFSM detailed for Activity EventListActivity. . . . .                                      | 49 |
| 5.4  | The third level of the HFSM for the application MyResults. . . . .   | 49 |





# List of Tables

|     |  |    |
|-----|--|----|
| 5.1 | Final list of Android applications to be tested. . . . .   | 46 |
| A.1 | Call Test Pattern results (FSM construction results) and number of activities explored dynamically by the <i>iMPAcT Tool</i> (with the <i>REiMPAcT</i> component). . . . . | 63 |
| A.2 | Percentage of activities explored by the <i>iMPAcT Tool</i> (with the <i>REiMPAcT</i> component). . . . .  | 64 |



# Abbreviations

|      |                                   |
|------|-----------------------------------|
| OS   | Operation System                  |
| UI   | User Interface                    |
| VM   | Virtual Machine                   |
| AUT  | Application Under Test            |
| API  | Application Programming Interface |
| FSM  | Finite State Machine              |
| ASM  | Abstract State Machine            |
| HFSM | Hierarchical Finite State Machine |
| GUI  | Graphic User Interface            |
| MBT  | Model Based Testing               |
| PBT  | Pattern Based Testing             |
| PBGT | Pattern Based GUI Testing         |
| API  | Application Programming Interface |
| DSL  | Domain Specific Language          |



# Chapter 1

## Introduction

Daily, thousands and thousands of mobile applications are built, and some of these applications end up becoming indispensable in our daily lives.

Mobile applications offer all kinds of service at a one-click distance, from bank transfers, online purchases, or even managing your email. These services are not centralized in a single application which leads to stores being flooded daily with new applications, which most often come to provide existing services. This growing number of applications available daily in stores is consequent to rapid development and technological stacks that make the development of mobile applications more accessible. Besides these technological factors, the relatively low cost of development has a high impact on these numbers.

In 2017, the number of people using a smartphone has exceeded 2.32 billion, and the combined sum of downloads from Apple's App Store and Google Play Store exceed 178.1 billion downloads, and these numbers for the year 2022 can reach a total value of 258.2 billion downloads [[Stad](#); [Stae](#)].

### 1.1 Problem

Most of the companies that develop mobile applications end up not spending much of their resources to ensure the quality of their products. This is the result of an expensive and time-consuming testing process where the tests are not automatically generated and executed. Besides these factors that make it difficult to test the mobile applications, there are other factors which aggravate this situation as, for example, the different OS where applications were developed, the various versions of the OS of the target devices, different screen sizes, different hardware, among others.

Mobile applications are becoming more popular and adopted, it is necessary to assure their quality and proper functioning. Although testing the applications does not guarantee 100% the nonexistence of errors, it allows to increase the confidence and security of the users when they use the application.

Taking into account all the difficulties associated with the test of mobile applications the most viable solution is the automation of the tests. It is in this context that the *iMPAcT Tool* appears.

The *iMPAcT Tool* uses Reverse Engineering processes and dynamic black box analysis to automate the testing of the recurring behavior (UI Patterns) presented on Android mobile applications. The *iMPAcT Tool* automatically explores the Android mobile application by firing UI events. After identifying the UI Patterns, the tool applies the correspondent Test Pattern.

Being a Black-Box tool, the *iMPAcT Tool* doesn't need to have access to the source code of the application to test it, besides, the fact that it is automatic saves time and manual labor. However, sometimes crawling techniques like this can not explore all the details needed and, it is necessary to adopt different solutions. The same happens to other Black Box testing techniques that sometimes cannot identify all the errors in the application and other techniques are adopted.

It is clear now that *iMPAcT Tool* has important features that can help the field of mobile applications testing, but it is also important to remember that this tool has problems that need to be corrected. The validation of this work involves improving some behaviors that are tested by the *iMPAcT Tool* and use the *REiMPAcT* approach to check if it is possible to extract the activities of an Android application dynamically.

## 1.2 Motivation and Goals

Currently, the market share of mobile users that have Android as their OS (Operation System) is 88%, so, the market for mobile applications is overwhelmed by Android applications [Stab].

There are already tools in the market that can help developers automate the testing process, as is the case of the framework developed by Google called UIAutomator. The solutions presented by these tools to date don't satisfy the users because the test sets still need to be handwritten by the developers.

What distinguishes the *iMPAcT Tool* from the other tools is the capacity to automate the test process, that is, both the test generation and the test execution are performed entirely by the *iMPAcT Tool*. This automation is only possible because the *iMPAcT Tool* has at its disposal a catalog of Patterns, and compares the Patterns identified in the AUT (Application Under Test) with the ones displayed in the catalog. Each UI Pattern presented in the catalog has a Test Pattern associated. After identifying the UI Pattern, the *iMPAcT tool* apply the respective Test Pattern, which basically tells the tool how to test that UI Pattern and how to check if the identified UI pattern is correctly implemented.

The main goal of this dissertation is improving the *iMPAcT Tool* in the following aspects:

- Improving the Call Test Pattern.
- Extend the *iMPAcT Tool* with reverse engineering capabilities.
- Extract the activity names of the Android applications dynamically.
- Build a hierarchical finite state machine (HFMSM) with three distinct levels of abstraction.

## 1.3 Methodology

The initial focus of this dissertation was the research about the *iMPAcT tool*, Mobile Testing, Related Approaches, Patterns, etc. After that, the implementation phase took place and, all details described. The final step was to verify if the objectives were achieved and present a set of experiments that validate the work done.

### Bibliographic Collection

Like other dissertations, the first thing to do is research on the State of the Art about topics related to the research work. In Chapter 2 some items are covered like, existent tools and current approaches for mobile testing, solutions for the problems of some approaches, model-based testing, pattern-based testing, reverse engineering techniques, modeling GUI, Android guidelines and conclusions.

### iMPAcT Tool

One of the critical factors of this dissertation is the *iMPAcT tool*. In Chapter 3 is presented a more in-depth analysis of this tool. This deeper analysis focuses on what is the background of the tool, which are the functionalities of the tool, what is different from the other tools and what contributions this tool has to offer to the world of Android mobile applications development.

### Implementation

After analyzing in more detail the characteristics of the *iMPAcT Tool* in Chapter 3, a description of all the work implemented and achieved throughout this dissertation is made. The main goal was to improve the Call Test Pattern and extend the *iMPAcT Tool* with reverse engineering capabilities to extract the activity names of the Android applications dynamically.

A description of the corrections performed in the Calls Test Pattern and a description of the new approach called *REiMPAcT* can be found in Chapter 4.

### Validation

To validate the work performed throughout this dissertation, several experiments were conducted and presented throughout Chapter 5. The main objective is to verify if it is possible to extract the name of the activities of the Android applications dynamically.

The first step of the validation process was to define the research questions that will be answered in Chapter 6. After that, the equipment used in the test environments is specified, as well as the criteria used to select the Android applications for testing and a list of the Android applications that will be used for the tests. Finally, the experiments and the respective results obtained are presented.

## Conclusions and Future Work

The conclusions of the experiments performed in Chapter 5 and future work can be found in Chapter 6.

### 1.4 Structure of the Dissertation

In addition to the introduction, this dissertation has five more chapters. In Chapter 2 is reported the State of the Art. In the Chapter 3 a deeper analysis of the *iMPAcT tool* is presented. In Chapter 4 is described all the details of the corrections performed in the Call Pattern of the *iMPAcT Tool* and the details of the implementations of the new approach called *REiMPAcT*. In Chapter 5 is presented the research questions and results of the experiments. Finally, the future work and conclusions can be found in chapter 6.



## Chapter 2

# State of the Art

This chapter is a review of the State of the Art in testing mobile applications and Reverse Engineering of mobile applications.

Section 2.1 presents the current state of the mobile applications market and focus areas for testing mobile applications.

There are several approaches to test mobile applications, so, Section 2.2 describes three Official frameworks and two non-Official frameworks to test mobile applications.

Section 2.3 presents the concept of Model-Based Testing for mobile applications, the main problems, and possible solutions to solve these problems.

Section 2.4 presents the concept of Pattern Based Testing (PBT) for mobile applications. Besides that, it presents a formal definition of a Pattern and is described the project behind the *iM-PACT Tool*, the Pattern Based GUI Testing (PBGT).

Section 2.5 presents several solutions and types of Software Reverse Engineering, with particular focus on Reverse Engineering of Android applications.

Section 2.6 presents several solutions to model Hierarchical GUIs and to diminish the problem of state explosion related to the state machines.

Section 2.7 presents the best practices for developing Android applications.

The conclusion and revision of the State of the Art are presented in Section 2.8.

### 2.1 The world of Mobile Applications

In 2017 approximately one-third of the world population (32.3%) owned a smartphone as we can see in Figure 2.1. Western Europe leads regional markets, with approximately 65% of its total population. Right after, North America has a penetration of around 64% of its total population. The Middle East and Africa end up being the smallest regional market with only 13.6% penetration of its total population [Staf].

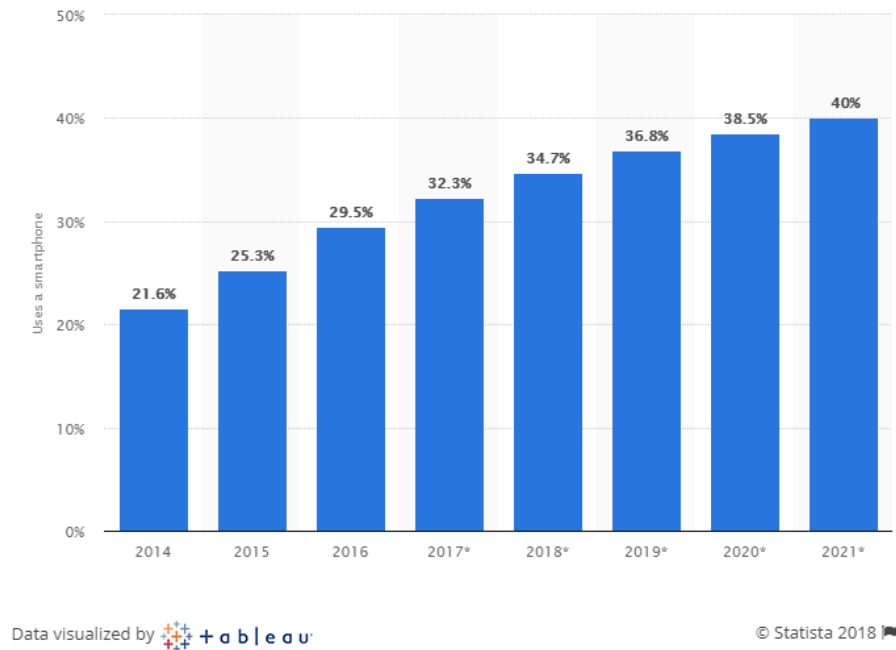


Figure 2.1: Smartphone user penetration as a percentage of the total global population from 2014 to 2021 [Staf].

Since 2009 the number of Android mobile applications available in the Google Store went from only 16 thousand to 2.6 million in September of 2018 as we can see in Figure 2.2. This growth happens because Android applications are increasingly cheaper and easier to develop [Stac].

In order to persist in this very competing market, it is necessary to present something original and fresh, with high quality and reliability. Quality and reliability is the key to success in the world of mobile applications. The quality and reliability can be ensured through testing processes. In 2017, 46% of the company’s pointed out the lack of adequate tools as the main challenge of mobile testing [Staa]. Since 2013 some focus areas for testing mobile applications have changed as we can see in Figure 2.3.

## 2.2 Current approaches to test Mobile applications

The automation of tests can be classified into two different sub-categories, test case generation, and test case execution. In the current state of Android mobile applications, the test case generation is the main focus. Some Official Android frameworks have the support to automate the test case execution. On the other hand, non-Official frameworks were created by the Android Community to full fill the necessities of the developers [MP15b]. A list of Official frameworks and Non-Official frameworks is presented in Section 2.2.1 and in Section 2.2.2, respectively.

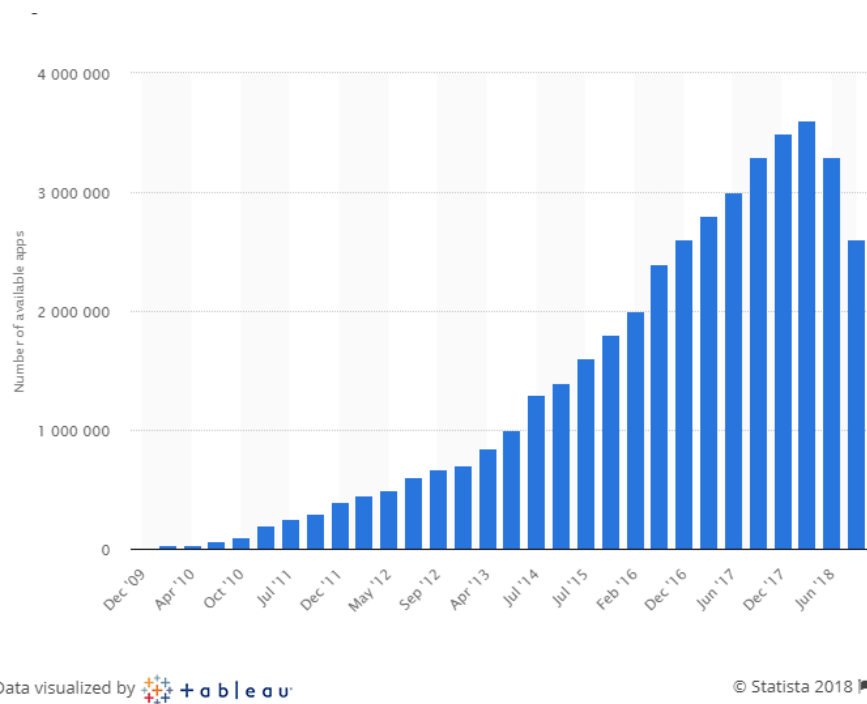


Figure 2.2: The number of available applications in the Google Play Store from December 2009 to September 2018 [Stac].

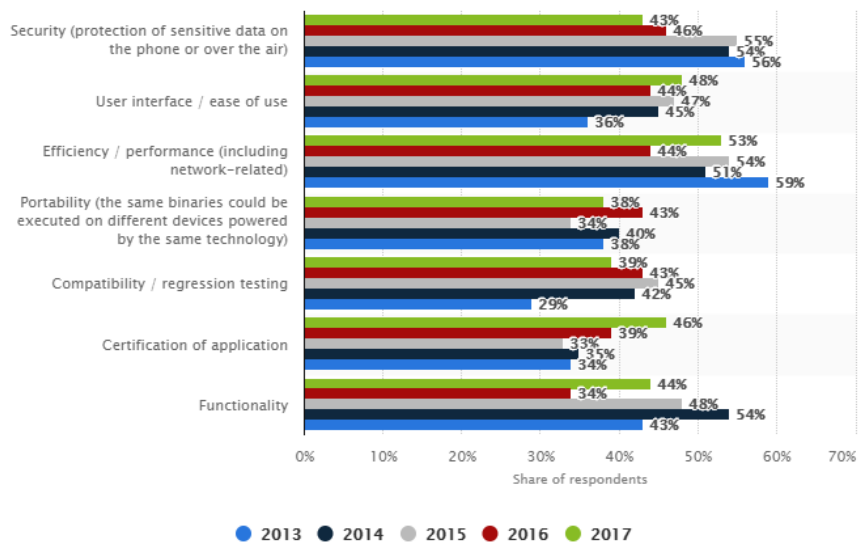


Figure 2.3: Focus areas for testing mobile applications from 2013 to 2017 [Staa].

## 2.2.1 Official Frameworks

### UI Automator

The *UI Automator* is a UI testing framework that offers a set of APIs in order to interact with the system and user applications. The APIs granted by the *UI Automator* supports operations like opening the settings menu or opening the application on the emulator or test device. One of the most important aspects of the *UI Automator* is the capacity that the framework already has for black box automated testing, *i.e.*, does not rely on the implementation of the target application [Fraf]. The *UI Automator* has three key features:

- **UI Automator viewer:** It is the component responsible for providing a GUI that scans and analyzes UI components on the Android device under test. It is also possible to inspect the layout hierarchy and obtain properties of the UI components. All this information will help to refine the test cases.
- **Accessing device state:** This component allows the framework to access device properties such as rotation, notifications, display size, sensors, etc.
- **UI Automator APIs:** It's a feature of the *UI Automator* that allows writing tests without knowing details about the implementation of the AUT.

### Espresso

*Espresso* is a framework developed by Google that provides a set of APIs to build UI tests based on the normal utilization of the application. Although it can execute black-box tests, the *Espresso* is better programmed to work with white-box testing style, that is, all the details of the AUT can be used to refine the tests [Frad]. The main APIs components of *Espresso* includes:

- **Espresso:** Is the start point to interactions.
- **ViewMatchers:** A flexible set of APIs that allows *Espresso* to view and check for adapter matching in the target applications.
- **ViewActions:** A set of APIs that allows *Espresso* to interact automatically with the UI and perform actions like click().
- **ViewAssertions:** A complex set of APIs that allows *Espresso* asserting the state of the current state of views.

### Calabash

*Calabash* is an open source and free acceptance testing framework developed and maintained by Xamarin [Cal]. *Calabash* allows users to write and execute automated acceptance tests for mobile applications, it has support for Android and iOS applications. *Calabash* also supports tests to be written using Cucumber [Cuc].

*Calabash* is composed of a set of libraries that enable test code to interact programmatically with native and hybrid applications. These interactions consist of end-users interactions, that can be:

- **Gestures** - Swipe, tap, rotate, etc.
- **Assertions** - For example, fill a login field.
- **Screenshots** - Screen dump the current view.

The *Calabash* framework provides real-time feedback and validations, which increase productivity.

### 2.2.2 Non-Official Frameworks

#### **Robotium**

Of all the tools, *Robotium* is the most popular Android testing framework among developers and Android Community. This popularity arises because its easy to write automatic black-box tests for Android applications, and it allows to write function, system, and user acceptance test scenarios [Frae]. Although its popularity, *Robotium* framework has advantages and disadvantages. The advantages of *Robotium* framework are:

- It has no minimum Android version to work and can test both hybrid and native Android applications.
- Does not need to access the source code because it uses a Black-Box testing style.
- Ability to modify sensors and states of running services on the device or emulator.

On the other hand, the disadvantages are:

- The interaction with the UI is limited, is not able to read the content of the screen.
- The framework can only access to services which the AUT has permission.

#### **Appium**

The *Appium* framework is an open-source tool that can be used by any mobile or web application. This framework uses as base other testing frameworks to automate their tests, for example, it uses *UI Automator* to test Android mobile applications and uses *Selenium Web-Driver* to develop tests for several programming languages such as Java, JavaScript, PHP, etc [Fraa]. The advantages of *Appium* framework are:

- Easy to use and setup, support several scripting languages.
- Support multiple platforms, it is possible to reuse the same test among all these platforms.

- Doesn't require access to the source code of the applications.

On the other hand, the disadvantages are:

- Technical issues, unexpected errors.
- Doesn't support image comparison.
- Unlike the *Robotium* framework, it only supports newer versions of the Android SDK because it depends on the *UI Automator* framework.

## 2.3 Model Based Testing for Mobile Application

Model-Based Testing (MBT) is a technique that enables the automation of test generation. This technique requires a model or specification of the system's behavior as input to generate the test cases. The output of MBT is a test suite [UL07].

The main issues of MBT are:

- The combinatorial explosion of test cases.
- The necessity of an input model of the application behavior. The model is built manually and is susceptible to errors.

To overcome the problems of the input models of the MBT, we can apply Reverse Engineering techniques to the application under test (AUT) or increase the level of abstraction of the model. The Reverse Engineering solution is addressed in Section 2.5. Also, in Section 2.4, a solution to diminish the effort required to build models is presented. The explosion of test cases is also a big problem for MBT, the focus on behavior patterns will help solve this problem.

## 2.4 Pattern Based Testing for Mobile Applications

This section presents several solutions that originated the Pattern-Based Testing (PBT) technique. One of the results of this technique is special because it is the reason behind the tool used in this dissertation, the *iMPACT Tool*. Section 2.4.1 explains the concepts of Patterns and what is the importance of Patterns to this work. Besides that, it identifies several PBT techniques. Finally, Section 2.4 describes the project behind the *iMPACT Tool*, the Pattern-Based GUI Testing (PBGT).

### 2.4.1 Patterns

In the last few years, some authors have investigated the benefit of Patterns for testing mobile applications.

The term "Pattern" is used to describe part of a system, and, at the same time, how to build that part of the system. Patterns usually describe pieces of the software used by designers, architects, and programmers in their systems [Ris98].

In 2009, Erik Nilson [Nil09] concluded that UI Design Patterns might be useful in resolving recurring problems in Android mobile applications development. The idea behind the solution is that if there is a behavior associated with a pattern, then it is possible to identify the pattern by identifying the behavior.

The author Eric Gamma [Ral02] represents the concept of pattern decomposed into four different main components:

- **Pattern name:** The primary purpose is to catalog the patterns, it is used to describe the design problem, the solution, and the consequences of both.
- **Problem:** Describes in which problems/context the pattern should be used.
- **Solution:** Describes the components that compose the design, the relation between them, responsibilities, etc. The solution presented is abstract to be used in any situation.
- **Consequence:** Describes the results of using the pattern and the impact of that pattern on the system. These two factors are essential for choosing the right pattern.

In order to simplify the definition, reuse, and addition of new patterns, a formal definition of these patterns is indispensable. The main goal of a pattern is testing recurrent behaviors. The formal definition of Pattern is defined as a set of tuples:

$\langle \text{Goal}, \mathbf{V}, \mathbf{A}, \mathbf{C}, \mathbf{P} \rangle$ , where:

- **Goal** is the ID of the pattern.
- **V** is the input of the pattern and corresponding values.
- **A** is the sequence of actions to perform in order to identify the presence of the pattern.
- **C** is a set of points to check if the pattern exists.
- **P** is the precondition that established the conditions in which pattern is applied.

For this dissertation, the main focus will be the UI Patterns and the Test Patterns. Taking into account the formal definition of Pattern presented before is possible to conclude that [MP15b; MP15c; MP15a]:

### UI Patterns

- **A** defines what actions to perform in order to verify the presence of the pattern.
- **C** validates the presence of the pattern.
- **P** defines when to verify if the pattern exists.

## Test Patterns

- **A** defines which actions the test perform.
- **C** indicates the result of the test.
- **P** defines in which conditions the test is applied.

### 2.4.2 Pattern Based GUI Testing

The Pattern-Based GUI Testing (PBGT) [MP14b] is a project that as the objective to reduce the effort needed to build a model for MBT.

The solution consists of presenting a modeling framework that is easy to use [MP13] and providing a Domain Specific Language (DSL) named **PARADIGM** [MP14a]. The objective of PARADIGM language is to increase the abstraction level of the model by describing the test goal instead of the system, allow relations between abstractions, and collect Test Patterns.

The UI Patterns presents common behaviors even when the implementation has small changes, the objective of this project is to create test strategies for different configurations so that in the ends it is possible to test different implementations of the same UI Pattern.

In the context of the PBGT was conducted a study to verify if this approach could also be applied to mobile applications. The study was a success and as a result emerged the necessity to develop test strategies to test mobile applications. The PBGT assume that GUIs based on the same UI Patterns should share the same UI Test strategy, this assumption result from the study referred before. Also, the concept of UI Test Pattern emerged from this project. The UI Test Patterns is the association of test strategies to the corresponding UI Pattern.

The tool implemented in the context of this project was developed using as a base the Eclipse Modelling Framework [Frac]. Four components compose this tool:

- An environment to build and configure GUI models, this environment can be shaped to the needs of each case.
- The tests models based on UI Test Patterns (UITPs), built using the PARADIGM language.
- Test case generation using PARADIGM models.
- A Reverse Engineering approach that automatically generates and extracts the model of a web application [SP14].

Although this project was initially projected to test web applications, the research and experiments on mobile applications concluded that it is possible to use this approach to test mobile applications. To use this approach is necessary to make some adaptations and develop specific test strategies. The reason behind the necessity to adapt some parts of the project is because of the technology stack and the nature of mobile applications that are different from the concept and



technology stack used in web applications [MP15c]. The gap between web and mobile applications in this context will open space for new tools, and that's where the *iMPACT Tool* comes from.

## 2.5 Software Reverse Engineering

Software Reverse Engineering is a field of Software Engineering that over the years, a lot of research has been done. Currently, there are several approaches to model desktop applications [RHR08; GPF10] and web applications [AFT09; MTR10].

### 2.5.1 Reverse Engineering approaches over the years

In 2003, Atif Memon presented the first approach for Reverse Engineering of desktop application User Interface (UI) called GUI Ripping [MBN03]. The GUI Ripping technique starts in any window of any Desktop application by detecting all widgets presented in the GUI, and after that, analyzes the application by executing the widgets detected. Atif Memon also states the importance of GUITAR framework [GUI] to capture models of application behavior and automatically generate test cases.

In 2005, Arie van Deursen and Elizabeth Burd [DB05] described Software Reverse Engineering as a process that "start with a low level representation of a system (such as binaries, plain source code, or execution traces), and try to distill more abstract representations from these (such as, for the examples just given, source code, architectural views, or use cases, respectively)".

In 2011, Cuixiong Hu presented a technique of Reverse Engineering that detects GUI bugs in Android mobile applications using Monkey [Mon; HN11]. The technique automatically generates test cases, creates random events, instruments virtual machines (VM), and produces a trace-file for detecting errors. The detection of errors is done after the execution.

In 2012, Inês Morgado and Ana Paiva presented a Reverse Engineering tool called *ReGUI* to extract a model from the execution of a Graphical User Interface (GUI) [CPP12]. The tool extracts the information from the GUI under analysis by a dynamic Reverse Engineering approach. This approach is independent of the programming language in which the GUI was written, broadening its applicability. This tool produces as output graphical representations of the GUI and a textual model in Spec# to be used in the context of MBT.

In 2012, Ali Mesbah presented a new Reverse Engineering technique for web applications [MDL12]. The crawling-based technique is to reverse engineering the navigation structure and paths of the web application under test. The approach presented by Ali Mesbah is called CRAWL-JAX [Frab] and automatically builds models of the web application GUI. The model is built by detecting the elements of the interface, after that executing them and finally comparing the states before and after the execution. This approach allows the automation of test cases generation and maintenance of web applications.

In 2014, Clara Sacramento and Ana Paiva presented a dynamic Reverse Engineering approach to extract UI Patterns from web applications [SP14]. The approach consists of extracting information from an execution trace and afterward inferring the existing UI Patterns and their configurations from that information. As a result, a model is created from this approach and can be used to generate test cases that are performed on the web applications.

### 2.5.2 Mobile Reverse Engineering

The main focus of this dissertation is on Reverse Engineering of Android mobile applications because the *iMPACT Tool* only works on Android devices, and there aren't many Reverse Engineering approaches to Android mobile applications.

It is possible to distinguish three different Reverse Engineering approaches:

- **Static Reverse Engineering:** This approach focus on how to automatically analyze the source code.
- **Dynamic Reverse Engineering:** This approach focus on how to analyze the application at run time.
- **Hybrid Reverse Engineering:** This approach combines the Static and Dynamic approach, bringing together the best of both worlds.

The main goal of Reverse Engineering of mobile applications is to obtain a model of the application behavior.

An example of a Hybrid approach of Reverse Engineering is presented in 2013 by W. Yang and T. Xie [YPX13]. Initially it is identified which are the possibles events to be executed, and after that, while exploring the application the events are executed and analyzed the effects on the application. The output is a model of the application.

## 2.6 Modeling Hierarchical GUIs

One of the solutions more appropriate and capable to model reactive systems are state machines. A state machine defines a set of states and transitions between states, where actions cause those transitions. GUIs are well known reactive systems that respond according to user actions. Regarding that information, state machines are useful to model GUIs and can be very useful in terms of testing of software applications [LY96].

One of the biggest problems associated with states machines is state explosion. One solution is to obtain an FSM from an Abstract State Machine (ASM) [Bör95].

HFSMs provide a solution to solve the state explosion problem. An HFSM is nothing more, nothing less than an FSM that vertices can represent a single state or group of states and the transitions between the states. The group of states and the transitions between them are FSMs. So, given an HFSM, it is possible to obtain an FSM by recursively substituting each group of states by its associated FSM.

HFSM is well-prepared to partition the behavior of a GUI. The hierarchical structure of the HFSM can imitate the hierarchical structure of objects and dialogues of the GUI. Consider an application with one main window, described by an FSM with  $m$  states, and  $k$  independent modal dialogs  $D_1, D_2, \dots, D_k$  that can be accessed from the main window, with each  $D_i$  described by an FSM with  $n_i$  states. If the application has no limitations, the total number of states of the application can be produced by  $m.n_1 \dots .n_k$ . But, if the  $D_i$ s are modal, only one dialog can be open at a time, so, fewer states have to be considered. If we consider the state machine that describes each dialog  $D_i$ , the possible states of the application can be grouped as:

- a group with  $m.1 \dots .1 = m$  states representing all dialogs closed and only the main window active.
- for each dialog  $D_i$ , a group with  $m.1 \dots (n_i - 1) \dots .1 = m(n_i - 1)$  states represents the situation where  $D_i$  is open and all the other dialogs are closed.

The total number of states of the application is  $m.(n_1 + \dots + n_k - k + 1)$ .

[Nik+05]

## 2.7 Android Best Practices

The best practices for developing Android applications are a set of documents provided by Google that contains the guidelines which should be followed by the developers to produce the best Android applications to the end user. This set of documents can be found at the Android Developers platform [Deva].

These guidelines are essential because they specify how the patterns and other aspects of the Android application should look and to behave in the final version of the application. They are used in the *iMPACT* Tool described in Chapter 3 as guidelines for testing the behavior of the Android application.

The set of documents are classified into three modules: Design, Develop, and Distribute. For this dissertation, the focus will be in the Develop module and partially in the Distribute module, although the Design module also affects the other modules in one way or another. The Develop module is divided into smallest modules with the best practices. These modules are:

- Best Practices for Interaction and Engagement
- Best Practices for User Interface
- Best Practices for Background Jobs
- Best Practices for Performance
- Best Practices for Security and Privacy
- Best Practices Permissions and Identities

From these modules, the dissertation focus on the best practices for Interaction & Engagement, User Interface and Background Jobs.

Regarding the Distribute module, on the Core App Quality, it is presented a list with the details on which the Android applications should be developed according to the Android standards [Qua].

In order to the Android applications become more familiar and friendly to the end user, Android has developed Material Design [And], which are a set of guidelines to help in the Android application design.

## 2.8 Conclusion

After the revision of the State of the Art, it is possible to conclude that this study area presents research maturity but at the same time offers a lot of space to grow and aspects to be improved.

The world of Smartphones and mobile applications has grown tremendously in the last few years, and more than ever, it is necessary to assure the quality and proper functioning of mobile applications. This involves testing mobile applications without spending too much time and resources. Throughout this chapter, several solutions were presented with the objective of satisfying the testing needs of mobile applications, all of them without great success because they fell short of expectations.

It is in this lack of suitable solutions to automate tests that the *iMPAcT Tool* emerges. The *iMPAcT Tool* automatically tests recurring behavior (UI patterns) present on Android mobile applications, *i.e.*, tests if the best practices in the development of Android mobile applications are followed. This tool is analyzed in more detail in Chapter 3.

In order to solve some of the problems related to Reverse Engineering approaches and software testing, arises the necessity of extending the *iMPAcT Tool* with Reverse Engineering capabilities through an approach called *REiMPAcT*. This approach is described in more detailed in Chapter 4.

## Chapter 3

# iMPAcT Tool

The *iMPAcT Tool* is presented as a solution to help the development of Android mobile applications, in the sense that verifies if the best practices for Android mobile development are met. The testing approach presented in this section by the *iMPAcT Tool* automates the testing of the recurring behavior (UI patterns) presented on Android mobile applications [MP19].

### 3.1 Approach

The *iMPAcT Tool* approach consists of exploring automatically Android mobile applications by firing UI events. After firing all the events, the *iMPAcT Tool* uses reverse engineering techniques to check if there are UI patterns present on the application under test (AUT). If a UI pattern is identified in the AUT, the *iMPAcT Tool* applies the corresponding Test Strategy, also known as Test Pattern. During reverse engineering the *iMPAcT Tool* uses a Catalog of Patterns that describes recurring behaviours, *i.e.*, UI Patterns. The *iMPAcT Tool* approach generates two types of output:

- **Report:** In the report is specified the matched UI Patterns, *i.e.*, which UI Patterns are present in the application and which of these patterns are correctly implemented or not in the AUT.
- **Finite State Machine:** The FSM represents the screens traversed during the exploration and how to navigate through them. Arrows and numbers identify the navigation through screens as we can see in Figure 3.1.

[MP15b]

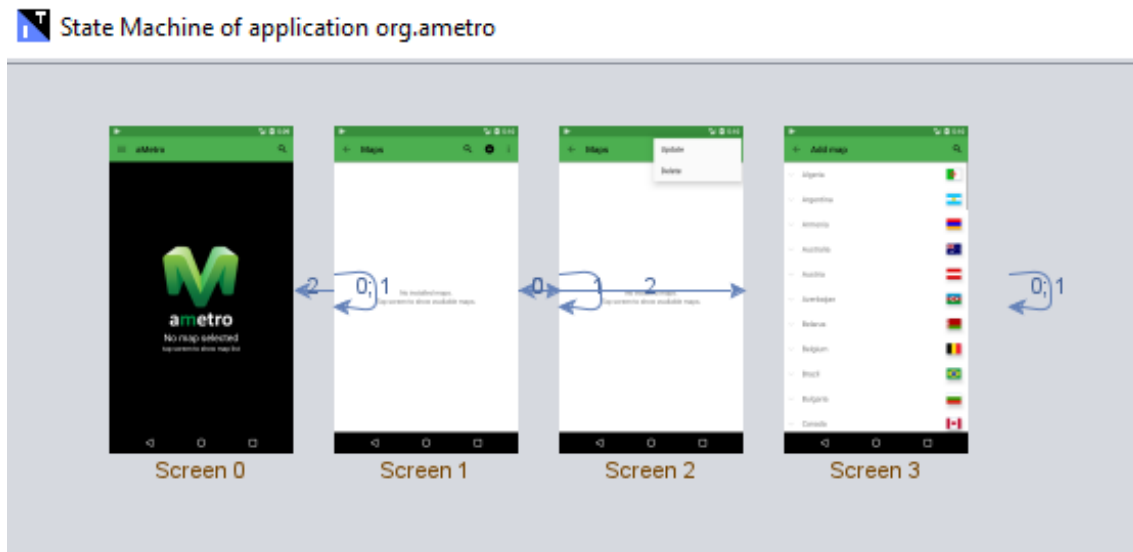


Figure 3.1: Finite State Machine of application org.ametro.

## 3.2 Implementation

The general implementation of the *iMPAcT Tool* approach can be described by the Algorithm 1 and by the Figure 3.2.

```

while (exploring) do
  | call fire_event;
  | call find_UI_Patterns;
  | if found_UI_Patterns then
  | | call apply_Test_Pattern;
  | | end
  | read exploring;
end

```

**Algorithm 1:** *iMPAcT Tool* execution algorithm.

### 3.2.1 Exploration

The exploration phase is responsible for analyzing the current state of the AUT and deciding which event to fire. The state of the application can be defined as the hierarchy of the elements present in the current screen, *i.e.*, each element of the screen is a node in a hierarchy tree.

Each node can be associated with an executable event. Currently, the *iMPAcT Tool* has the ability to fire events like click, long click, edit, and check. Although the tool has the ability to fire other events, they are only used in the testing phase.

After identifying the possible events that can be executed, the *iMPAcT Tool* prioritizes them:

1. Events not executed from a list.
2. Events not executed except clicking in the *Up Bottom* that takes the AUT to a previous state.
3. Events already executed that somehow lead to screens with events still to run.
4. Click on the *Up Bottom*.

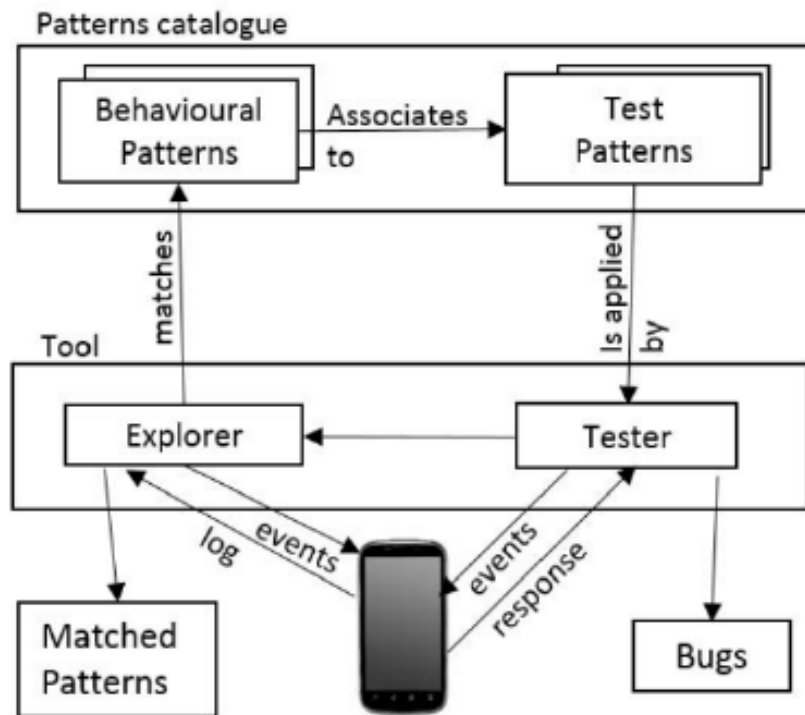


Figure 3.2: *iMPAcT Tool* approach [MPF14].

If there is no more available events, the *back* button is pressed.

The exploration ends when it reaches the home screen. The stop condition of the exploration process is reaching the home screen, that can be forced by pressing the *home* button [MP15b; MP15a].

### 3.2.2 Patterns matching

Patterns matching takes place after an event is fired. The *iMPAcT Tool* tries to identify which UI Pattern are present on the AUT. A UI Pattern is only considered found if the *preconditions* and all *checks* are met [MP15b; MP15a].

### 3.2.3 Testing

After detecting a UI Pattern, the corresponding Test Pattern are applied. Applying a Test Pattern implies verify if all *preconditions* are met and execute all the necessary *actions* to verify the *checks*

to decide if the test passes or fail. In the end a report is generated containing the final verdict, *i.e.*, the test results [MP15b; MP15a].

### 3.2.4 iMPAcT Tool configuration

The *iMPAcT Tool* is composed by two main components, developed in *java* [CP17].

- **iMPAcT Tool Installer:** The installer is responsible for implementing the interface of the *iMPAcT Tool*. This interface is where the test process is configured and where the user enters the information about the Android application to be tested.
- **iMPAcT Tool Tester:** The tester is the *back-end* of the *iMPAcT Tool* and is responsible for implementing the Exploring, Patterns matching and Testing.

To start the test, the user needs to complete the missing information on the interface. The interface of the *iMPAcT Tool* can be seen in Figure 3.3.

In this interface, the user can:

- Indicate the Android application to be tested, by providing the path to the APK or the *package* name.
- Choose the type of exploration.
- Where to run the test, emulator or device.
- Choose the patterns to be tested. The *iMPAcT Tool* provides a Pattern catalog in the interface.

### 3.2.5 Types of exploration of the iMPAcT Tool

As mentioned above, the user will need to choose a type of exploration before starting the test [CP17]. The *iMPAcT Tool* has four different types of exploration:

- **ONLY\_ONCE:** It only triggers each available event one time, *i.e.*, only triggers events that haven't been triggered yet.
- **PRIORITY\_TO\_NOT\_EXECUTED:** It is possible to repeat events, but only after all the events of a screen are triggered, and that event is necessary to access another screen.
- **MORE\_COMPLEX\_PRIORITY:** Events associated with elements present in lists have priority over the rest. It only accesses another screen after triggering all the events.
- **ALL\_EVENTS:** The events aren't differentiated. This can lead to that not all events being triggered.



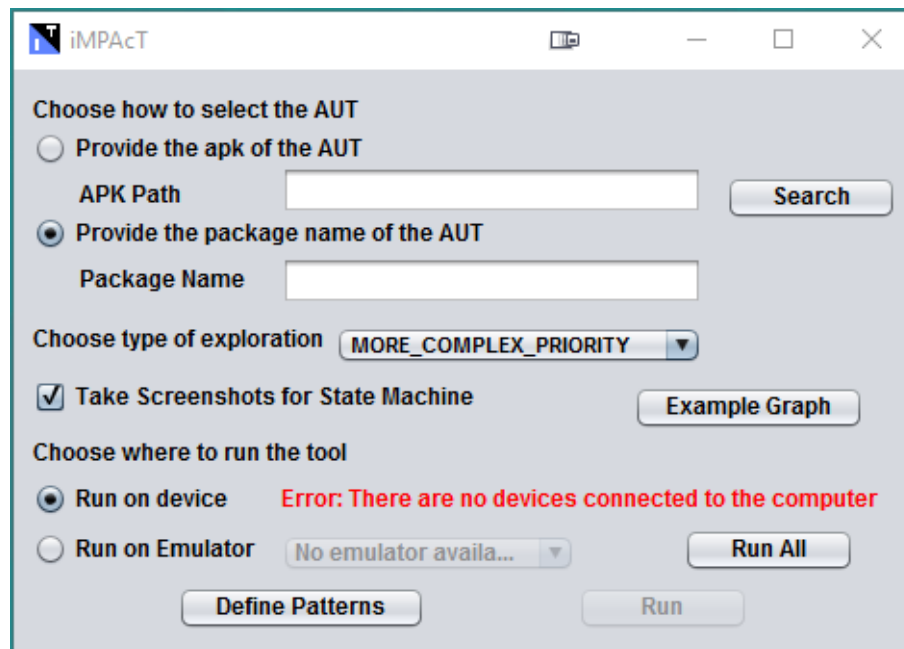


Figure 3.3: The interface of the *iMPAcT Tool*.

### 3.3 Patterns Catalog

One of the most important aspects of the *iMPAcT Tool* is the catalog of patterns because it defines how the test will be executed, and to which UI patterns that test will be applied. The catalog contains a set of UI Patterns and the corresponding Test Pattern. The first concepts of UI Pattern and Test Pattern are described in Section 2.4.1, where a Pattern was defined as:

$\langle \text{Goal}, \mathbf{V}, \mathbf{A}, \mathbf{C}, \mathbf{P} \rangle$ , where:

- **Goal** is the ID of the pattern.
- **V** is the input of the pattern and corresponding values.
- **A** is the sequence of actions to perform in order to identify the presence of the pattern.
- **C** is a set of points to check if the pattern exists.
- **P** is the precondition that established the conditions in which pattern is applied.

Along this section is described how the UI Patterns and the Test Patterns were implemented on the *iMPAcT Tool*. The development and implementation of these Patterns were based on Android guidelines to design [Des] and test of Android mobile applications [Tes]. Currently, the *iMPAcT Tool* only implements Patterns classified as "Visual design and user interaction" and "Functionality" by the Android Guidelines [Gui; CP18].

### 3.3.1 Side Drawer Pattern

Android mobile applications have several forms of hierarchical navigation through screens. One of the most known is the *Side Drawer*, also known as *Navigation Drawer* [Dra] UI Pattern. There are two ways to open a *Side Drawer*, the first consists of swiping the screen from the left edge to the center and the second consists of clicking on the App button (normally located on the left of the *Action Bar*). An example of a *Side Drawer* is presented in Figure 3.4.

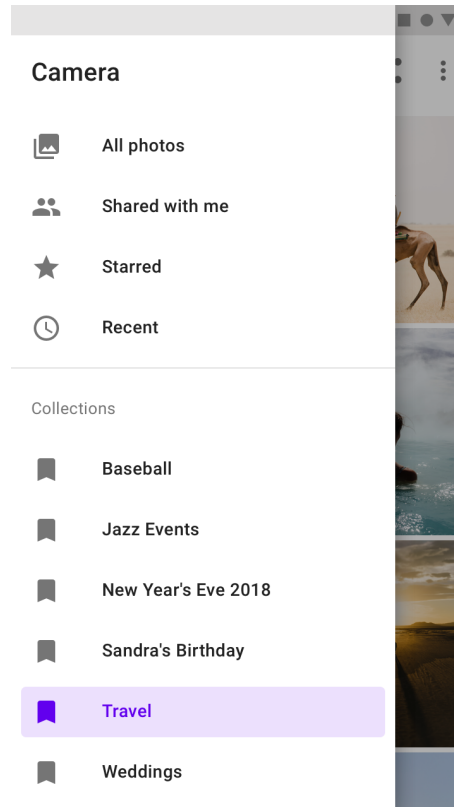


Figure 3.4: Example of a *Side Drawer* pattern [Dra].

To identify the presence of a *Side Drawer* a UI Pattern was defined:

- Goal: "Exists a Side Drawer"
- V: {}
- A: [read screen]
- C: {"Side Drawer exists but is not visible"}
- P: {"true"}

To check if the *Side Drawer* UI Pattern is correctly implemented, *i.e.*, if the *Side Drawer* reaches the full height of the screen a Test Pattern was defined:

- Goal: "Side Drawer reaches the full height of the screen"
- V: {}
- A: [read screen, open side drawer, read screen]
- C: {"reaches the full height of the screen"}
- P: {"UI Pattern identified && side drawer available && Test Pattern not applied yet to the current Activity"}

### 3.3.2 Orientation Pattern

The Android mobile devices have two possible orientations: portrait and landscape, as we can see in Figure 3.5.

When rotation occurs on the device, the application follows the movement and also updates the screen according to the position of the device. According to the Android Guidelines for testing [Gui], it is important to focus on two main aspects:

- No user *input* should be lost, all the information entered by the user should remain after the rotation.
- The *widgets* should remain in the screen after the rotation.



(b) Landscape orientation

(a) Portrait orientation

Figure 3.5: Example of possible orientations.

The UI Pattern that verifies if it is possible to change the orientation of the screen is defined as:

- Goal: "Rotation is possible"
- V: {}
- A: []
- C: {"it is possible to rotate the screen"}
- P: {"true"}

The *orientation* UI Pattern has two different Test Patterns. The first Test Pattern is applied when the UI pattern is identified, and it has not been tested yet in the current activity. The main objective of this first test is to rotate the screen and verify if the *widgets* remained on the screen. If initially there are no *widgets*, the screen is *scrolled* to make the last verification and carry on with the test. The first Test Pattern is defined as:

- Goal: "Main components of the UI are still present"
- V: {}
- A: [read screen, rotate screen, read screen, scroll screen, read screen]
- C: {"UI main components still present"}
- P: {"UI Pattern is present && Test Pattern not applied on current activity"}

The second Test Pattern is applied when the user entered information, and the pattern has not been applied to an element that has changed. The orientation of the screen is switched and is verified if the entered data remains. The second Test Pattern is defined as:

- Goal: "Data unchanged when screen rotates"
- V: {}
- A: [read screen, rotate screen, read screen, scroll screen, read screen]
- C: {"data entered by the user was not lost"}
- P: {"UI Pattern is present && user entered data && Test Pattern not applied on current activity"}

### 3.3.3 Resource Dependency Pattern

Most Android mobile applications use external resources, such as *GPS*, *NFC*, or *Wifi*. Some of these applications are dependent on those resources. Because these applications rely on these resources, it is necessary to ensure that applications do not crash when those resources are no longer available, as is indicated in the Android Guideline [Gui].

The UI Pattern is defined as:

- Goal: "Resource in use"
- V: {"resource", resource\_name }
- A: [read resource status]
- C: {"resource is being used by the application" }
- P: {"true" }

The corresponding test strategy, *i.e.*, Test Pattern is defined as:

- Goal: "When resources are no longer available, the application does not crash"
- V: {"resource", resource\_name }
- A: [read screen, turn off resource, read screen]
- C: {"application didn't crash" }
- P: {"UI Pattern && Test Pattern not applied on current activity" }

### 3.3.4 Tabs Pattern

In several Android mobile applications is possible to find *tabs* such Instagram, Facebook or even Skype. *Tabs* allows organizing content across different screens, data and other interactions [Tab]. An example of a *tab* is presented in Figure 3.6.

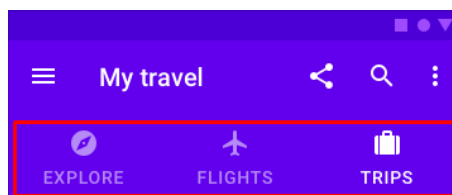


Figure 3.6: Example of a *tab* [Tab].

There are some guides about how to use and which are the common behaviors of *tabs* [BA]:

1. In order to make the UI easier to navigate, there should be only one set of *tabs*.

2. In Android mobile applications the *tabs* should be displayed at the upper part of the screen.
3. The action of horizontally scrolling a *widget* should change the selected *tab*.

The UI Pattern is defined as:

- Goal: "Presence of *Tabs*"
- V: {}
- A: [read screen]
- C: {"There are tabs present"}
- P: {"true"}

The corresponding test strategy, *i.e.*, Test Pattern is defined as:

- Goal: "Only one set of *tabs*"
- V: {}
- A: [read screen]
- C: {"there is only one set of *tabs* at the same time"}
- P: {"UI Pattern && Test Pattern not applied on current activity"}

and

- Goal: "*Tabs* position"
- V: {}
- A: [read screen]
- C: {"*tabs* are in the upper part of the screen"}
- P: {"UI Pattern && Test Pattern not applied on current activity"}

and

- Goal: "Horizontally scrolling a *widget* should change the selected *tab*"
- V: {}
- A: [read screen]
- C: {"the selected tab changed"}
- P: {"UI Pattern && Test Pattern not applied on current activity"}

### 3.3.5 Back Pattern

In all Android devices, there is a button called *back* which allows the user to visit all the screen visited previously (Android *back stack* [TS]), regardless of the state. In most cases, the *back* button is handled and managed by the Android system itself, but it is possible to manually configure the behavior of this button to meet the needs of a given Android mobile application. This pattern executes three important steps:

1. Verifies if the AUT uses a behavior defined by the android system and not a custom.
2. Verifies if the AUT is not on the home screen because it would return the user to the main screen of the device and activate the stopping condition of the tool.
3. Verifies if by clicking on the *back* button the user is navigated to the previous screen.

Given that the *back* button is integrated into the Android system, it is safe to assume that it is always present, so it is not necessary to formalize a UI Pattern to identify it. An example of the *back* button is presented in Figure 3.7.



Figure 3.7: Example of a *back* button.

The corresponding test strategy, *i.e.*, Test Pattern is defined as:

- Goal: "*back* button navigates to the previous screen"
- V: {}
- A: [read screen, press *back* button, read screen]
- C: {"the screen was changed to the previous" }
- P: {"Not the first screen visited && Test Pattern not applied on current activity" }

### 3.3.6 Background Pattern

In Android mobile applications, it is normal to have tasks to run in parallel. The Android system itself offers support to run these tasks in the *background*. Besides that, the Android system provides the user the ability to send an application to *background* by pressing the *home* button [Nag]. This action causes the application to stop being used, but it is expected that the application will continue to run in the background and that at any moment it is available and that the information is not lost.

There is no way to formalize this UI Pattern because the only condition to the test strategy is that the application under test needs to be open. An example of the several applications running in *background* is presented in Figure 3.8.

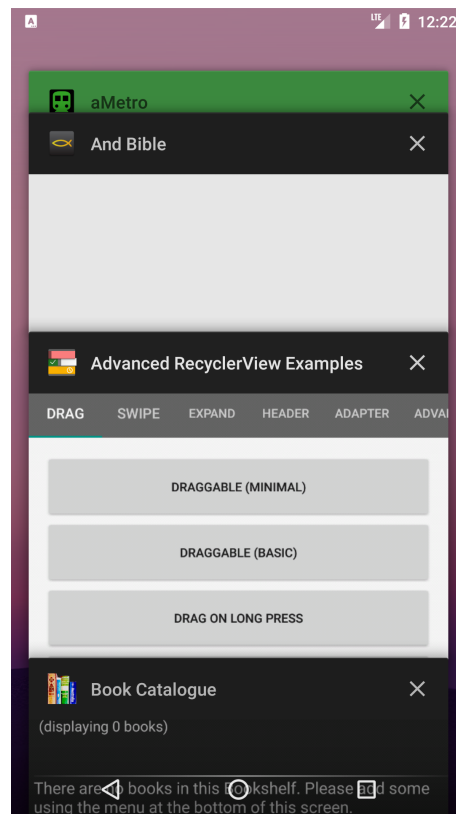


Figure 3.8: Example of several applications running in *background*.

The corresponding test strategy, *i.e.*, Test Pattern is defined as:

- Goal: "The application is sent to *background* and there is no data loss"
- V: {}
- A: [read screen, send application to *background*, read screen, send application to *foreground*, read screen]
- C: {"the status of the application is the same before && after being in *background*" }
- P: {"UI Pattern identified && Test Pattern not applied on current activity" }

### 3.3.7 Up Pattern

In the world of Android mobile application development it is a good practice to add a *Up* button in the *action bar* of the application whose function is to navigate to the screen that is logically related in the hierarchy of the application screens [Nav]. This pattern aims to verify if in each different screen there is a navigation *Up button*, with the exception of there being a *side drawer*. Furthermore, it also checks if after clicking on the button, the navigation is performed according to the hierarchy of the application screens.



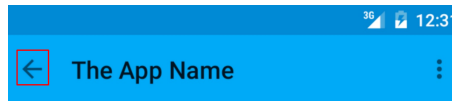


Figure 3.9: Example of the *Up* button.

An example of the *Up* button is presented in Figure 3.9.

The UI Pattern is defined as:

- Goal: "Exists *Up* Pattern"
- V: {}
- A: []
- C: {"Has *action bar* && does not exist *side drawer*"}
- P: {"Application is not in the home screen"}

The corresponding test strategy, *i.e.*, Test Pattern is defined as:

- Goal: "Exists *Up* button and navigate to parent screen"
- V: {}
- A: [read screen, click on *Up* button, read screen]
- C: {"application goes to the logical parent screen"}
- P: {"UI Pattern present && Test Pattern not applied on current activity"}

### 3.3.8 Action bar Pattern

According to the Android development guide [Devb], the use of *action bars* is a good practice to promote consistency between applications, making Android mobile applications more similar, this results in less time for the user becoming familiar with new applications. The *action bar* has three main features:

1. Place holder to assign an identity to the application and place the user in it.
2. Provides access to the user to important features of the application.
3. It supports *navigation* and *applications* view changes.

In addition to the features listed above, the *action bar* has some elements that are usually present. Usually, the *action bar* is composed by the name of the application on the left side, a floating menu on the right side and the *Up* button on the left side of the title, but only when the application is not in the home menu. An example of an *action bar* is presented in Figure 3.10.



Figure 3.10: Example of an *action bar*.

As with other patterns, there is no way to formally verify the existence of this UI Pattern because the *action bar* will always be present in any Android mobile application.

The corresponding test strategy, *i.e.*, Test Pattern is defined as:

- Goal: "*Action bar* exist in the application"
- V: {}
- A: [read screen]
- C: {"is *action bar* present"}
- P: {"UI Pattern present && Test Pattern not applied on current activity"}

### 3.3.9 Calls Pattern

The Android phones are smartphones but at the same time are usual cell phones that can make and receive calls from other cell phones. It is also essential to keep in mind that a smartphone allows the user to use several applications on his device. With this, it is necessary to ensure that these two features work correctly, so when a smartphone receives a call in the middle of an interaction with an application, the device should be able to return to the same state of the application after hanging up the call. An example of an incoming call state while using an application can be seen in Figure 3.11.

The corresponding test strategy, *i.e.*, Test Pattern is defined as:

- Goal: "Verifies if the application works properly after an incoming call"
- V: {}
- A: [read screen, hang up the call, read screen]
- C: {"the application didn't crash && the states of the application before and after the call are the same"}
- P: {"incoming call"}

## 3.4 Visualization of the results

The *iMPAcT Tool* generates two artifacts as output when the exploration cycle ends. These artifacts allow the user to understand the application AUT better, the actions taken, and the results of the test. The two generated artifacts are:

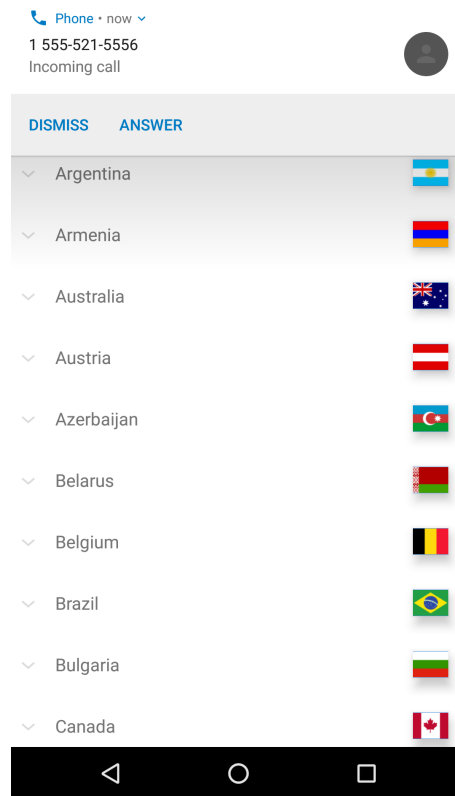


Figure 3.11: Example of incoming call state while using an Android application.

- Report of the exploration and test.
- Finite state machine of the Android application.

### 3.4.1 Report

The final report of the *iMPACT Tool* is composed by the exploration log and the test results.

#### 3.4.1.1 Exploration Log

The exploration log is composed of the identification of the application under test, the exploration parameters, and the sequence of events launched during exploration.

**Identification of the application under test:** The application under test is identified by the name of the package.

**Identification of the exploration parameters:**

1. Android version of the emulator/device where the application under test is running;
2. Screen height and length;
3. The y-coordinate from where the application began to be drawn. Unless the application is in *full screen*.

### Identification of the sequence of events launched during exploration

Each one of the events identified by the *iMPAcT Tool* is defined as a tuple  $\langle \text{Event}, \text{Element}, \text{Executions} \rangle$ , where:

1. **Event:** Identifies the type of launched event and the entry value for that event. The entry value is not necessary for all events.
2. **Element:** Identifies the events launched in an *element* of the application screen. Is defined by the tuple  $\langle \text{class}, \text{resource id}, \text{text}, \text{content description}, \text{margins} \rangle$ , where:
  - *class*: identifies the type of element.
  - *resource id*: identifies the element. This property is neither unique nor mandatory.
  - *text*: text to be displayed in the screen, within the boundaries of the element. This property is not mandatory.
  - *content description*: text associated with an element to describe the result of a click on that element, but the results it's not represented on the screen. This property is not mandatory.
  - *margins*: indicates the position of an element on the screen. This property is mandatory.
3. **Executions:** Records the number of times that a certain event was launched in a given element.

#### 3.4.1.2 Test Results

The test results contain:

- The number of realized and identified events, as the corresponding percentage.
- The Patterns identified, and in the cases of being identified if they were correctly implemented or not.
- Duration of the exploration.

The test results report allows users to manage and consult all the information relative to the tests, all the steps of the exploration cycle, all the identified patterns, and the corresponding results of each pattern.

#### 3.4.2 Finite state machine

The *iMPAcT Tool* uses reverse engineering processes to obtain models of the application GUI. This model is described in a Finite State Machine where each state of the fork corresponds to different activities from the application under test, on the other hand, each transaction represents the events that cause the transition from one screen to another.

It is possible to distinguish three types of *states* in the FSM:

1. **AUT Screen:** The AUT Screen is identified by Screen\_<*i*>, where *i* is the id of the screen. This id is attributed whenever a new screen is detected during the exploration cycle. Screen\_0 is the initial state of the application.
2. **crash Screen:** Whenever a failure occurs in the application it is registered as a *crash Screen* and represented in the FSM. There is only one node of this type, all events that originate failures are connected to that node.
3. **out of AUT Screen:** Whenever the exploration cycle exits the application it is registered as a *out of AUT Screen* in the FSM. There is only one node of this type, all the events that exit the application are connected to that node.

### 3.5 Conclusion

After a more in-depth analysis of the tool, it is possible to conclude that the *iMPAcT Tool* is innovative because it combines reverse engineering techniques with automatic identification of UI Pattern, and it has a solid base for future upgrades and reworks. Also, the *iMPAcT Tool* is easy to use and fully automated, the user doesn't need to have any previous knowledge about the tool, or the Android application that will be tested. That's why the *iMPAcT Tool* will revolutionize the market.



## Chapter 4

# Implementation

Throughout this chapter will be described and discussed the modifications implemented on the *iMPAcT Tool* in the context of this dissertation. In Section 4.1 will be explained all the changes performed on the Call Pattern already implemented on the *iMPAcT Tool*. Following, in Section 4.2 will be described the new approach called *REiMPAcT*, as well as the entire process to achieve the final solution. Finally, in Section 4.3 will be presented some conclusions of the work performed.

The experiments performed in the work developed and described in this chapter will be specified and described in more detail in Chapter 5.

### 4.1 Patterns Catalogue

In this section will be reviewed the Call Pattern present in the *iMPAcT Tool* Patterns Catalog. After a review of the *iMPAcT Tool*, it was possible to verify that the output of the tool (FSM) for the Call Pattern was not the desired one, so it was necessary to fix the problem identified in this pattern.

#### 4.1.1 Call Pattern

The Call Pattern structure consists of two main components, Call Activity and Caller Detection. The Call Activity component is responsible for making calls, while the Call Detector component is responsible for detecting incoming calls. The Call Pattern requires two emulators to perform the tests, the first one is responsible for receiving the calls and executing the *iMPAcT Tool*, and the second is responsible for making the calls. By default, the *iMPAcT Tool* assumes that the first emulator to be started is the primary one.

After initialization of the emulators, the test can finally be started. At the same time that the calls are received, it is possible to test other patterns on that emulator. Calls are denied using a technique called Java Reflection.

The Call Pattern check if: 1) the call was rejected, 2) if the application interacted correctly with the system and 3) if it doesn't fail, it compares the screen before and after the call rejection.

#### 4.1.1.1 Problem

After a more in-depth analysis of the *iMPAcT Tool*, and its outputs, it was possible to visualize that the finite state machine (FSM) generated after testing the Call Pattern was wrong. The FSM screens often didn't belong to the tested Android application, or when they belonged, they were not the correct ones. To find out the source of the problem, more exhaustive tests were performed on the Call Pattern.

In order to build the FSM, it is necessary to have access to the screenshots taken during the exploration, the `gson.txt` file, and the `impactTool_log.txt` file, all these files are stored in the emulator memory, and at the end of the *iMPAcT Tool* exploration cycle is imported into the tool folder on the computer.

It was possible to verify that the files saved in the emulator were correct, just weren't being extracted correctly to the tool folder, because the shell command to extract the files from the emulator was not specifying the right emulator to extract the file.

#### 4.1.1.2 Implemented Solution

After finding the source of the problem, it was necessary to find a solution to specify the correct emulator on the shell command that extracts the files from the emulator. It is important to remember that the primary emulator is the first to be open. To find out the name of this emulator was used the ADB command:

```
1 $ adb shell devices
```

With the help of the shell command presented before, it was possible to discover that the primary emulator is the emulator-5554. To specify the emulator in shell command responsible for extracting the files just was added the following line of code in the command:

```
1 $ adb -s emulator-5554
```

## 4.2 REiMPAcT

In Chapters 1 and 2, it was possible to verify that the market of Android applications presents a high growth rate and, consequently there are more and more Android applications with different functionalities.

Given that the applications are increasingly sophisticated and have more functionalities, it is necessary to carry out tests to ensure quality and proper functioning. However, it is essential to remember that the complexity of applications makes testing more difficult and time-consuming. Test automation is a solution to these problems because it automatically detects failures on the application, but doesn't always help to find the source of the failures.



To better understand the application under analysis and find the source of its failures was developed the *REiMPAcT* approach. This approach extends the *iMPAcT Tool* with reverse engineering capabilities and is able to extract information about the activities and the states traversed by a dynamic exploration of an Android application. In addition to extracting information about activities, this approach creates a navigation map of the explored activities, checks the navigation flow of the applications, and helps in software testing context.

#### 4.2.1 REiMPAcT architecture

The *REiMPAcT* approach is composed by two main components, as we can see in the architecture diagram in Figure 4.1.

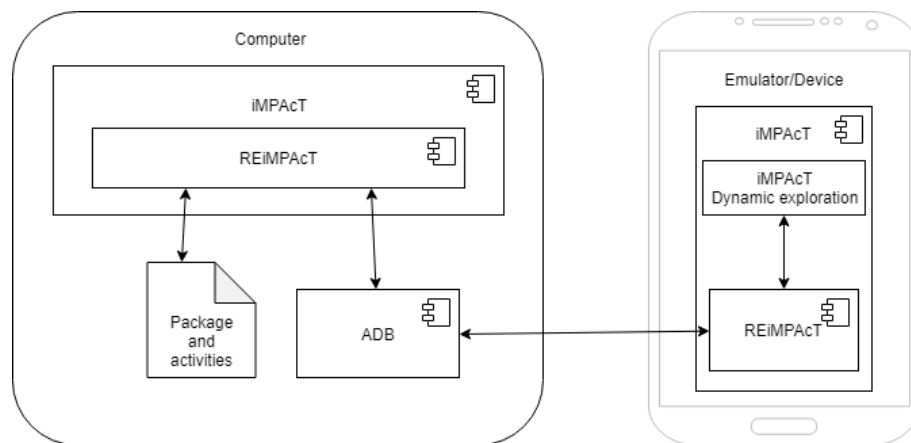


Figure 4.1: The architecture of the reverse engineering approach (*REiMPAcT*).

The *REiMPAcT* component running on the Computer is called *REiMPAcT* Activities Finder and the *REiMPAcT* component running on the Emulator/Device is called *REiMPAcT* Time Synchronizer.

##### 4.2.1.1 REiMPAcT Time Synchronizer

This component of the *REiMPAcT* approach is responsible for extracting and saving the exact moment at which each screenshot was taken.

Throughout the *iMPAcT Tool* dynamic exploration, several screenshots are taken along the traversed activities. In order to be able to associate each screenshot with its respective activity, the *REiMPAcT* approach collects the screenshots taken and several metrics. The collected metrics are composed by the exact moment each screenshot is taken (using Java Date library) and the actions that lead to the change of screen, *i.e.*, the change of state.

All of these metrics are stored in a text file in the Emulator, at the end of the *iMPAcT Tool* dynamic exploration cycle, both the text file and the screenshots are exported to the *iMPAcT Tool* project folder located in the computer.

#### 4.2.1.2 REiMPAcT Activities Finder

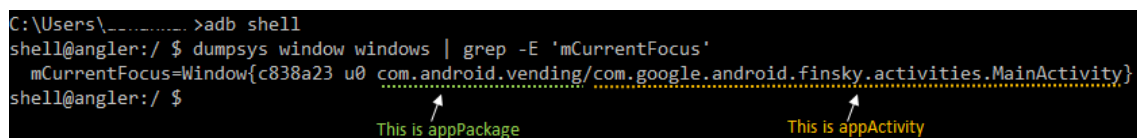
This component of the *REiMPAcT* approach is responsible for extracting the name of the activities traversed during the dynamic exploration of the *iMPAcT Tool*.

The extraction of the activities names traversed is conducted in parallel with the normal flow of execution of the *iMPAcT Tool* dynamic exploration component installed in the emulator or device.

The *REiMPAcT* Activities Finder component runs a process on the *iMPAcT Tool* component present in the computer, which checks the activity name of the application every second. This time interval can be customized.

To do this, the *REiMPAcT* approach uses the ADB (Android Debug Bridge) command line tool that allows communicating with a device or emulator. The command used outputs the package name and the name of the current activity of the application under analysis at that moment, as it can be seen in Figure 4.2.

```
1 $ adb shell
2 $ dumpsys window windows | grep -E "mCurrentFocus"
```



The screenshot shows a terminal window with the following text:
 

```
C:\Users\... >adb shell
shell@angler:/ $ dumpsys window windows | grep -E 'mCurrentFocus'
mCurrentFocus=Window{c838a23 u0 com.android.vending/com.google.android.finsky.activities.MainActivity}
shell@angler:/ $
```

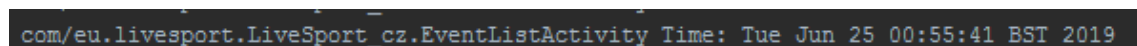
 Two green arrows point to the package name 'com.android.vending/com.google.android.finsky.activities.MainActivity' in the output line. The first arrow points to 'com.android.vending' and is labeled 'This is appPackage'. The second arrow points to 'com.google.android.finsky.activities.MainActivity' and is labeled 'This is appActivity'.

Figure 4.2: ADB Shell command output.

All the information collected along the execution of the *REiMPAcT* Activities Finder component is stored in a text file, including the names of the activities and the exact time in which this information was collected (using Java Date library).

When the *iMPAcT Tool* exploration cycle ends, all the information collected by the two components (Time Synchronizer and Activities Finder) is available for use, *i.e.*, all the text files with the pretended information are in the project folder. Given that it is collected and stored the exact moment at which each screenshot is taken and, at the same time it is collected and stored the name of each activity and the exact moment when this information was collected it is possible to associate each activity with a screenshot by comparing the moments when both information was collected.

For example, by carefully analyzing the Figures 4.3 and 4.4, it is possible to conclude that at the exact moment when the Android application was in the *EventListActivity* activity the screenshot number 3 was taken.



The screenshot shows a single line of text:
 

```
com/eu.livesport.LiveSport_cz.EventListActivity Time: Tue Jun 25 00:55:41 BST 2019
```

Figure 4.3: Output of the Activities Finder component from the *REiMPAcT* approach.

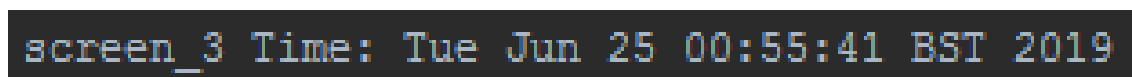


Figure 4.4: Output of the Time Synchronizer component from the *REiMPAcT* approach.

After associating all this information, it is possible to build the HFSM (which is constructed and displayed using a Java library called *mxGraph*).

### 4.2.2 *REiMPAcT* output

The output of the *REiMPAcT* approach is an HFSM (Hierarchical Finite State Machine) composed by three distinct levels of abstraction as we can see in Figure 4.5.

At the First Level of abstraction, it is represented all the activities traversed by the dynamic exploration algorithm of the *iMPAcT Tool*. At this level, it is only necessary to identify the activities explored and the user actions that traverses the activities.

The navigation within activities is represented by arrows and user actions. An arrow between two activities represents user actions that allow navigating from the origin activity to the destination activity, *i.e.*, actions that departure from a state of the origin activity and reaches a state of the destination activity.

At the Second level of abstraction, it is represented, for each activity, the screens traversed and the actions that allow that sequence of screens exploration. At this level, it is necessary to associate each screenshot to an activity and, for the activity in focus identify the user actions that traverse screens of that activity. This level of the HFSM represents the inner navigation for each identified activity of the explored Android application.

At the Third level of abstraction, it is represented all the screens traversed during the exploration of the *iMPAcT Tool*. This level is built using an already existing component of the *iMPAcT Tool* that provides an infrastructure to build a finite state machine (FSM) of the application under test (AUT). At this level, it is necessary to identify the screenshots and the sequence of actions that traverses screens in order to build the FSM. This abstraction level of the HFSM displays the general navigation within the screens of the AUT.

Consider an Android application with a set of different screens,

$$S = \{s_1, \dots, s_{|S|}\} \quad (4.1)$$

different activities,

$$A = \{a_1, \dots, a_{|S|}\} \quad (4.2)$$

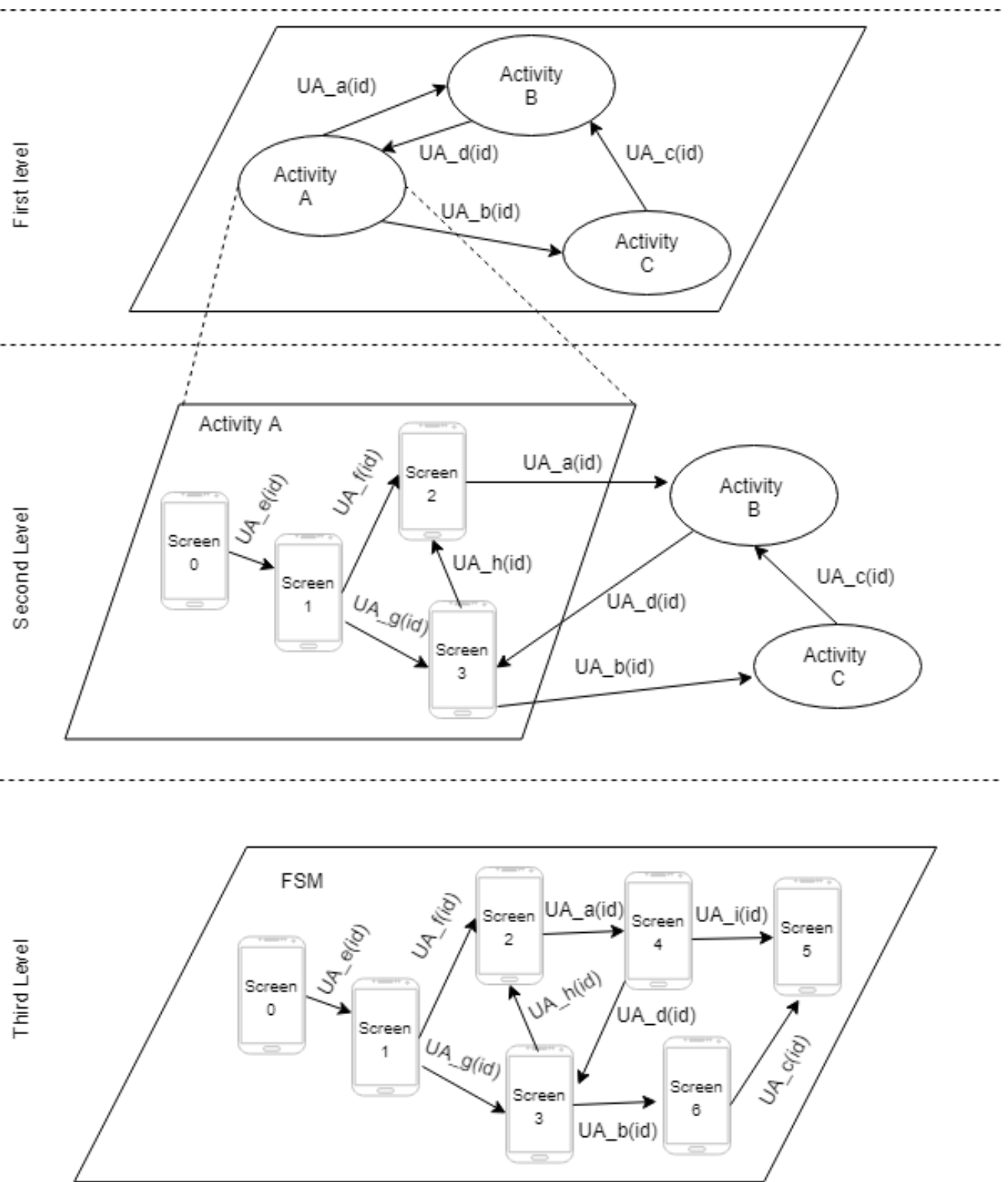


Figure 4.5: Hierarchical Finite State Machine (HFSM) with 2nd level detailed for Activity A.

and the set of possible user actions over that Android application.

$$UA = \{ua_1, \dots, ua_{|UA|}\} \quad (4.3)$$

A navigation path is a sequence of possible user actions that traverse the screens of the Android application.

$$N = [(s_o, x, s_d) | s_o, s_d \in S, x \in UA] \quad (4.4)$$

The First level of the HFSM abstracts the states of the Android application. It displays the activities and the user actions which traverse the origin state and the destination state that belongs to another activity. It may be obtained by:

$$[(a1, x, a2) | a1, a2 \in A \wedge \exists n \in elems(N) | n.\#1 \in a1 \wedge n.\#3 \in a2 \wedge n.\#2 = x \wedge a1 \neq a2] \quad (4.5)$$

Now, consider  $M$  as the set of all possible paths of type  $N$ . This set of possible paths is the complete behavior of the Android application.

The automatic exploration process of the *iMPAcT Tool* traverses,  $T$ , which is a subset of  $M$ .

$$T \subset M \quad (4.6)$$

The paths within  $T$  explore a subset of states in  $S$ .

The states of the Android application belong to activities, which means that we may consider each activity  $a \in A$  as a subset of  $S$ .

$$a \subset S \quad (4.7)$$

So, when we project  $T$  over an activity  $a \in A$  we get all the actions (i.e., transitions) performed over states of activity  $a$ . The Second Level of abstraction of the HFSM can be obtained by:

$$proj(T)_a \quad (4.8)$$

The Third Level of abstraction of the HFSM can be obtained by the full set of:

$$(s_o, x, s_d) | s_o, s_d \in S, x \in UA \quad (4.9)$$

### 4.3 Conclusions

Reviewing the work developed throughout this dissertation, it was possible to fix the problems of the previously implemented Call Pattern and extend the *iMPAcT Tool* with reverse engineering capabilities through an approach called *REiMPAcT*. The correction performed in the Call Pattern at FSM level adds value to the tool, as it is now possible to obtain the correct FSM in all patterns available in the Pattern Catalog of the *iMPAcT Tool*. As for the extension of the *iMPAcT Tool* with Reverse Engineering capabilities, it adds value to the tool as it allows to analyze in more detail the Android application under test through an HFSM with three distinct levels of abstraction and a set of collected metrics that are extremely useful in the context of software testing.

# Chapter 5

## Validation

Throughout this chapter will be presented the results of the experiments performed in the Call Pattern and in the *REiMPAcT* approach to validate the corrections and improvements implemented in the *iMPAcT Tool*.

The research questions are presented in Section 5.1. On Section 5.2, it is displayed the technical specifications of the device, computer, emulator, and environment of the experiments.

The test methodology, the Android applications requirements and, the final list of Android applications to test can be found in Section 5.3. The results of the experiments performed can be seen in Sections 5.4, 5.5 and 5.6.

Finally, the answers to the research questions and the conclusions can be found in Chapter 6.

### 5.1 Research Questions

In this section, it is presented the research questions raised during the development of this dissertation. The objective of the research questions is that, after the experiments carried out throughout this chapter it is possible to have a clearer idea of the results obtained and conclude whether the work done contributed or not to the improvement of the *iMPAcT Tool*.

- **RQ 1:** Is the *iMPAcT Tool* able to build an FSM of an Android application when testing the Call Pattern?
- **RQ 2:** Is it possible to extract the name of the activities of an Android application by a dynamic process?

- **RQ 3:** Which is the percentage of activities explored dynamically within a limited time period?
- **RQ 4:** What is the additional percentage of activities explored when exploration time increases?

The answer to these questions will be answered in Chapter 6, according to the experiments results that are described in this chapter.

## 5.2 Technical Specifications

In order to guarantee the consistency of the obtained results, experiments were carried out on different devices. That is, in addition to the experiments performed in real devices, there is also the need to perform experiences in emulated devices. The devices have the following technical specifications:

### LG G3

- Operation System: Android Nougat (7.0)
- CPU: Quad-core 2.5 GHz Krait 400
- Chipset: Qualcomm MSM8974AC Snapdragon 801
- GPU: Adreno 330
- RAM: 3GB

### Custom Desktop

- Operation System: Windows 10 Pro x64
- CPU: Intel Core i7-8700K Hexa-Core 4.9GHz
- Chipset: Gigabyte Z370 AORUS Gaming 3
- GPU: MSI GeForce GTX 1060 Gaming X 6GB GDDR5
- RAM: 16 GB 3200Mhz
- Android Emulator: Nexus 6 API 24
- Emulator Version: Android Nougat (7.0)

In the case of the emulator, is specified the technical specifications of the hardware on which it is running.



## 5.3 Test Methodology

This section describes the test methodology used in the experiments to ensure that the results obtained are reliable. To achieve these results is prepared a test plan and a list of Android applications to run the tests.

The following steps compose the test plan:

1. Select Android applications to be tested by the *iMPAcT Tool*.
2. Run the *iMPAcT Tool* for the Call Test Pattern over each Android application.
3. Collect the test results, *i.e.*, check the FSM construction.
4. Run the *iMPAcT Tool* (with *REiMPAcT component*) during a limited time period of 5 and 15 minutes for each Android application.
5. Collect the results of the tests, *i.e.*, the number of activities explored.
6. Perform a manual inspection on the Android Manifest file of each Android application.
7. Calculate the percentage of activities explored for each experiment on each Android application.
8. Establish the final conclusions of the results obtained from the entire process.

The first step of the test plan was to make a consistent and weighted selection of Android applications. To ensure that, the Android applications have been selected from different categories available on the Google Play Store. The variety of categories allows obtaining a set of Android applications with distinct functionalities to be tested. In addition to the categories, there are other important requirements to choose Android applications for these experiments:

- Must be compatible with the specified devices.
- Be free.
- Be a native Android application.
- Not rely on other applications.
- Not require a login to interact with the application.
- Not require authorization access to the device contacts, camera or documents.
- Have more than five thousand downloads in the Google Play Store.
- Must be readable by the *UiAutomator*.

| Application         | Version         | Category             |
|---------------------|-----------------|----------------------|
| aMetro              | 2.0.1.7         | Navigation and Maps  |
| Paris Metro         | 1.1.19          | Navigation and Maps  |
| EasyBus Porto       | 2.0.10          | Navigation and Maps  |
| Forest Fires        | 2.6.1           | News and magazines   |
| Portugal Newspapers | 4.0.3           | News and magazines   |
| Sapo Newspaper      | 3.1.1           | News and magazines   |
| Google News         | 5.11.0.19041218 | News and magazines   |
| Math tricks         | 2.28            | Education            |
| Pilot               | 1.1.0           | Education            |
| Periodic Table      | 0.1.83          | Education            |
| MyResults           | 3.2.0           | Sports               |
| Hockey              | 3.1.0           | Sports               |
| Monefy              | 1.9.4           | Finances             |
| 1Money              | 2.1.1           | Finances             |
| Home Workout        | 1.2.1           | Fitness and Health   |
| Pedometer           | 1.0.43          | Fitness and Health   |
| Wysa                | 0.9.5           | Medical care         |
| Moodpath            | 2.1.2           | Medical care         |
| Wikipedia           | 2.7.50282       | Books and references |
| Poems               | 1.9             | Books and references |

Table 5.1: Final list of Android applications to be tested.

The final list of Android applications to be tested was obtained from the entire possible set by random choice and can be found in Table 5.1.

It is essential that the results obtained are as reliable as possible, so it is necessary to keep in mind that the order in which the events are fired during the exploration are random and it can influence the test results. To reduce the random factor associated with the *iMPAcT Tool* exploration, each experiment was repeated three times, and the result is the average of those results.

The exploration algorithm chosen was *Priority to Not Executed* taking into account its features that allow repeating some events that may have already executed and thereby obtain the largest number of events fired and the largest number of screens traversed.

## 5.4 Finite State Machine

The purpose of this section is to conduct an experiment that will help answer the first research question (**RQ 1:** Is the *iMPAcT Tool* able to build an FSM of an Android application when testing the Call Pattern?). This experiment consists of executing the Call Test Pattern for 5 minutes for each Android application in Table 5.1.

At the end of each test, a static analysis of the FSM is performed to verify if it is correct, *i.e.*, if the screens sequence is right and if the screens belong to the tested Android application.

After performing this experiment, it should be possible to determine if the *iMPAcT Tool* can build an FSM of an Android application when testing the Call Pattern.

Given that it wouldn't be practical to display the FSM generated after testing the Call Pattern for the 20 applications, only one example is presented in Figure 5.1.

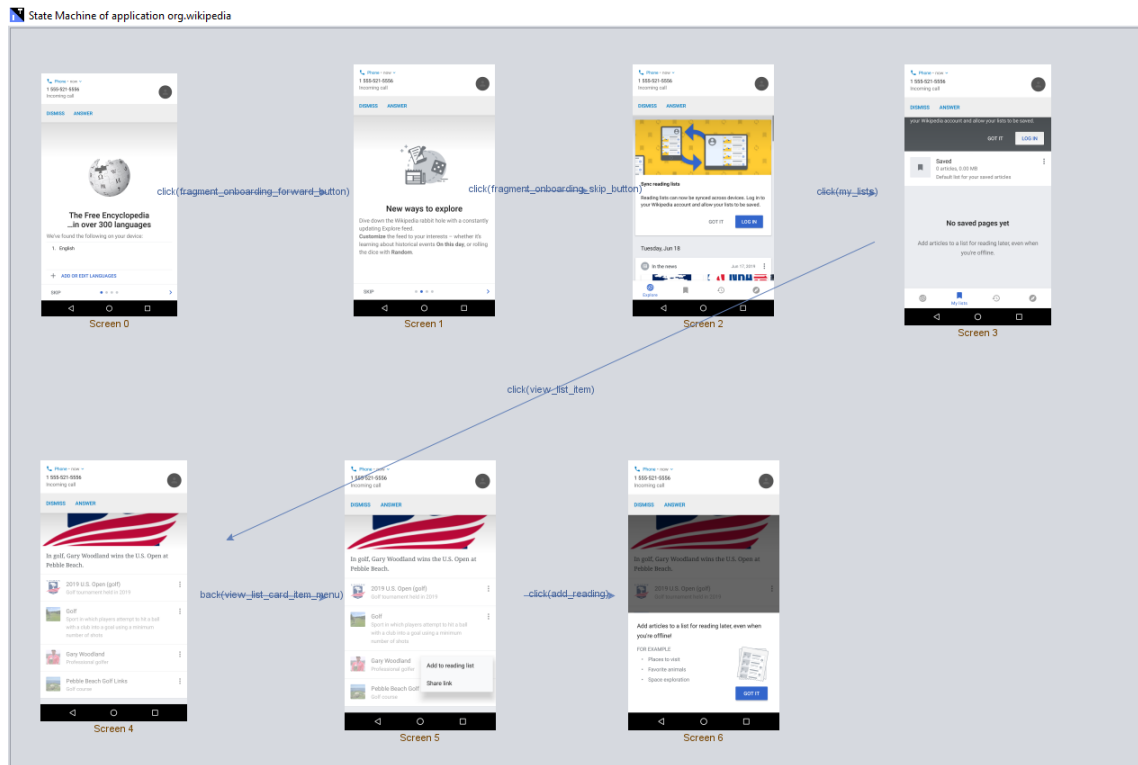


Figure 5.1: Call Test Pattern output (FSM) for the application Wikipedia.

After analyzing the results achieved (Table A.1) we can conclude that it was possible to construct the FSM of all the Android applications tested, *i.e.*, of the 20 Android applications tested, it was possible to build the FSM in 20 of the cases (100%).

## 5.5 Extracting Activities Dynamically

The purpose of this section is to conduct an experiment that will help answer the second research question (**RQ 2**: Is it possible to extract the name of the activities of an Android application by a dynamic process?). This experiment consists of executing for 5 minutes the *iMPAcT Tool* (with *REiMPAcT* component) over the Android applications of the Table 5.1 and collect the number of activities explored.

After performing this experiment, it should be possible to determine if it is possible to extract the name of the activities of an Android application by a dynamic process.

It wouldn't be a good practice to present the *iMPAcT Tool* output for the 20 applications, so it is only presented an example of the output for the Android application MyResults.

The Figure 5.2 presents the first level of the HFSM, the Figure 5.3 presents the second level (for the activity `EventListActivity` of the Android application) and Figure 5.4 presents the third level of the HFSM.

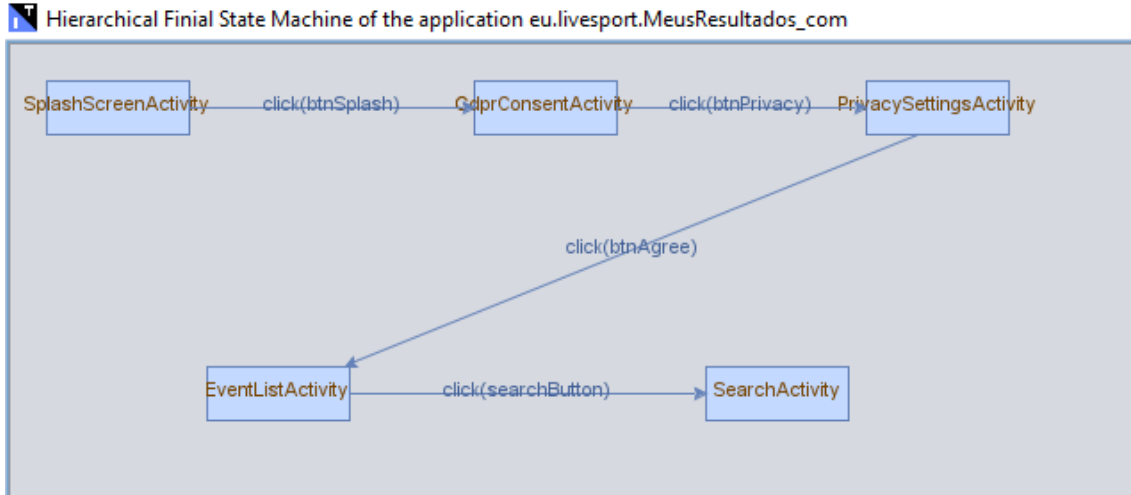


Figure 5.2: The first level of the HFSM for the application MyResults.

After careful analyses of the Figures 5.2, 5.3 and 5.4 presented before, it is possible to verify that:

1. During the dynamic exploration, it was possible to extract five distinct activities (*SplashScreenActivity*, *GdprConsentActivity*, *PrivacySettingsActivity*, *EventListActivity* and *SearchActivity*) in a period of 5 minutes.
2. The *EventListActivity* activity is composed of Screen\_3 and Screen\_4.
3. The dynamic exploration captured seven different screenshots.

After analyzing the results achieved (Table A.1) we can conclude that it is possible to extract the name of the activities of an Android application by a dynamic process, *i.e.*, of the 20 Android applications tested, it was possible to extract the names of the activities explored in 20 of the cases (100%).

## 5.6 Percentage of Activities Explored

The purpose of this section is to conduct an experiment that will help answer the third research question (**RQ 3:** Which is the percentage of activities explored dynamically within a limited time period?). This experiment consists of executing for 5 minutes the *iMPACT Tool* (with *REiMPACT* component) over the Android applications of the Table 5.1 and collect the number of activities explored to calculate the percentage of activities explored.

In order to calculate the percentage of activities explored, it is necessary to identify the total number of activities of the Android application under analysis and the number of activities explored.



Figure 5.3: The second level of the HFSM detailed for Activity EventListActivity.

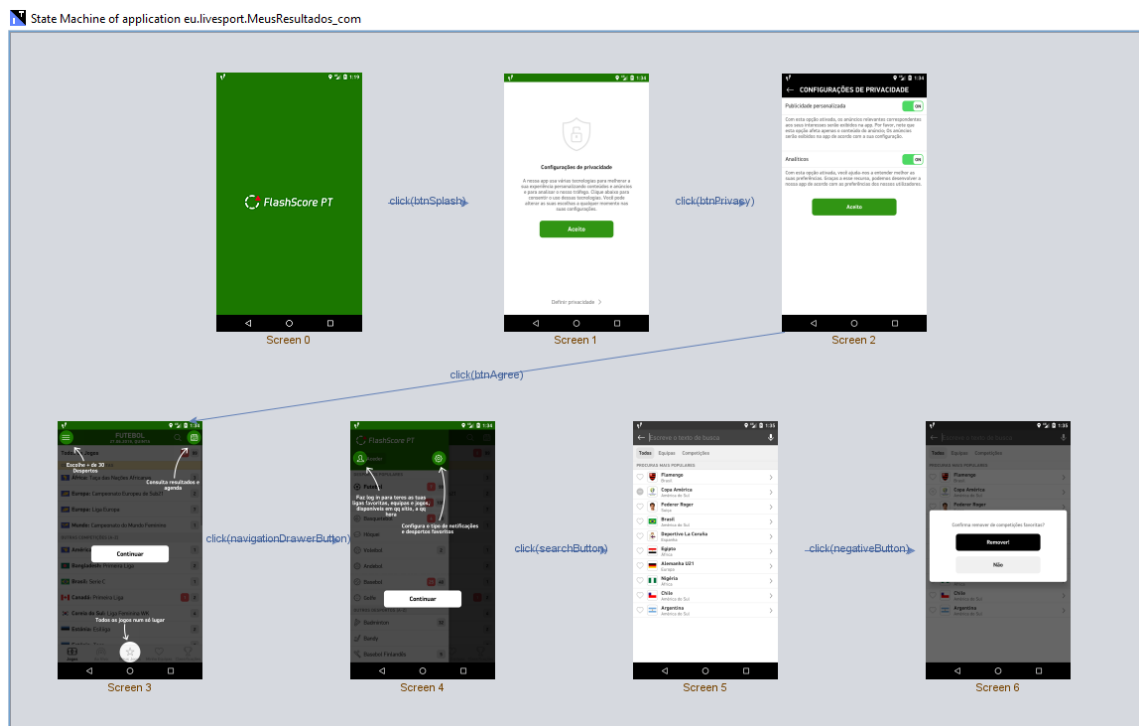


Figure 5.4: The third level of the HFSM for the application MyResults.

With the *REiMPAcT* approach, it is possible to obtain the number of activities explored of an Android application, but it isn't possible to obtain the total number of activities of that same Android application. For that, it is necessary to perform a static analysis over the source code, or the APK of the Android application. Not all the applications from Table 5.1 are open-source, so it is necessary to parse the APK.

Several platforms allow downloading the Android application APK by the package name that can be obtained in the Google Play Store. After obtaining the APK of each application in Table 5.1, it is possible to use the Analyze APK functionality of the Android Studio to analyze the Android application and get the `AndroidManifest.xml`.

In the `AndroidManifest.xml` are defined all the activities of the Android application, so it is possible to know how many activities there are in the Android application and with that to know the information that was lacking to calculating the percentage of activities explored.

After performing this experiment, it should be possible to determine the percentage of activities explored dynamically within a limited time period (5min).

The chosen exploration algorithm (*Priority to Not Executed*) may influence the results achieved by the *REiMPAcT* approach. If the *iMPAcT* Dynamic exploration isn't able to exercise the complete behavior of the application under analysis, the *REiMPAcT* approach will not be able to detect all the activities and extract the complete HFSM of the Android application.

Some activities declared in the `AndroidManifest.xml` are not related to the screens of the Android application. The *REiMPAcT* approach will not detect these activities because it only identifies the activities focused on the device or emulator. Excluding the activities that aren't related to the screens, the percentage of activities explored would be higher.

After analyzing the results achieved (Table A.2) we can conclude that the worst result belongs to the Google News Android application that only explored 11% of the activities in the experiment limited to 5 minutes. For the same experiment, the best result belongs to the Forest Fires application that explored 67% of the activities. The average percentage of activities explored in the 5-minute limited experiment is approximately 30% of the activities.

## 5.7 Exploration Time and Percentage of Activities Explored

The purpose of this section is to conduct an experiment that will help answer the fourth and last research question (**RQ 4:** What is the additional percentage of activities explored when exploration time increases?). This experiment consists of increasing the execution time of the previous experiment (Section 5.6) to 15 minutes and collect the number of activities explored to calculate the percentage of activities explored.

One of the most important aspects of the *REiMPAcT* approach is checking if the time of the experiments is directly related to the number of explored activities, this is why the experiment time is increased to 15 minutes.

After analyzing the results achieved (Table A.2) we can conclude that 18 of the 20 applications were able to explore more activities with the increase of the experiment time to 15 minutes, and the remaining 2 Android applications explored the same percentage of activities. Compared to the limited experiment of 5 minutes, no application explored fewer activities in the 15-minute experiment. The average percentage of activities explored in the 15-minutes experiment is approximately 39% of the activities.

## 5.8 Conclusions

Throughout this chapter was presented four different experiments. The first experiment was intended to verify if the *iMPAcT Tool* is able to build the FSM of an Android application when testing the Call Pattern, and, so answer the first research question. After the first experiment, the second experiment was carried out to answer the second research question. To do this, the *iMPAcT Tool* (with *REiMPAcT* component) is executed for 5 minutes for each Android application and it is verified if it is possible to extract the name of the activities through a dynamic process. Completed the second experiment, the third experiment takes place to answer the third research question. To do this, the *iMPAcT Tool* (with *REiMPAcT* component) is executed for 5 minutes for each Android application and is calculated the percentage of activities explored within a limited time period. Finally, the fourth experiment takes place and aims to answer the fourth and last research question. Regarding that, the execution time of the *iMPAcT Tool* (with *REiMPAcT* component) is increased to 15 minutes and is calculated the percentage of activities explored to check if the time is directly related to the number of activities dynamically explored. All the results of these experiments can be found in Appendix A.

All the answers to the questions raised during the execution of this dissertation can be found in Chapter 6. In addition to the answers for the research questions, it is also possible to find future work and conclusions regarding the work developed throughout this dissertation.





## Chapter 6

# Conclusions and Future Work

In Chapter 5, several questions were raised with the purpose of validating the work developed in the context of this dissertation.

Throughout this chapter, the questions raised in the previous chapter are answered.

To conclude the dissertation, a summary of all the items addressed throughout the dissertation is presented.

### 6.1 Discussion

#### 6.1.1 RQ 1: Is the *iMPAcT Tool* able to build an FSM of an Android application when testing the Call Pattern?

To answer this question was carried out an experiment to demonstrate the capability of the *iMPAcT Tool* to build an FSM of an Android application when testing the Call Pattern. From the experiment performed in Section 5.4 we can concluded that the modifications performed in the *iMPAcT Tool* were a success, *i.e.*, the *iMPAcT Tool* is now able to build an FSM of an Android application when testing the Call Pattern.

#### 6.1.2 RQ 2: Is it possible to extract the name of the activities of an Android application by a dynamic process?

This question is answered based on the experiment performed in Section 5.5. After analyzing the results obtained, we can conclude that it is possible to extract the name of the activities of an Android application by a dynamic process.

### 6.1.3 RQ 3: Which is the percentage of activities explored dynamically within a limited time period?

In order to answer this question, it is performed an experiment in Section 5.6. Regarding the results obtained, we can conclude that for a limited experiment of 5 minutes the average percentage of explored activities is 30%.

### 6.1.4 RQ 4: What is the additional percentage of activities explored when exploration time increases?

To provide an answer to the last question was performed an experiment discussed in Section 5.7. By analyzing the results obtained, we can conclude that the time of exploration influences the number of activities explored, *i.e.* if we increase the exploration time, we are able to detect and extract more activities in 90% of the cases. In this experiment, the average percentage of explored activities increased approximately 9% compared to the experiment limited to 5 minutes.

## 6.2 Future Work

In order to improve the *iMPAcT Tool* in the future, the following two aspects should be considered:

- **Change the current approach to associate the screens to the correct activity:** The current approach uses time units to synchronize the information regarding the screens and their activities, which is susceptible to error. It is important to remember that the synchronization of the activities with the screens is one of the most important aspects for the construction of the second level of abstraction of HFSM and in case of an error the HFSM of this level is incorrect. To reduce the probability of these types of errors occurring should be adopted an approach that collects the activity information at the same time and in the same component responsible for taking the screenshots.
- **Export the HFSM to PDF:** At this moment, the HFSM generated by the *REiMPAcT* approach is lost after closing the interface of the *iMPAcT Tool*. To improve the visualization of the results of the *REiMPAcT* approach, it would be relevant to export all the levels of abstraction of the HFSM to PDF in order to store all this information and access it whenever needed.

## 6.3 Conclusions

After the work performed on this dissertation, it is visible the dimension and the impact in the industry caused by the testing field and mobile applications, especially the Android applications that were the focus of this dissertation. The mobile applications market is moving more and more people and money.

The entire field of testing and mobile applications is a vast universe and, therefore, it is impossible to take into account all aspects related to these topics. Daily we are confronted with many changes and innovations in this area, so it is required much research and adaptation to keep up with the latest trends of tests, devices, approaches, etc.

The solutions presented throughout this dissertation aim to correct some problems related to the *iMPAcT Tool* (Call Pattern) and also increase the tool's Reverse Engineering capabilities through an approach called *REiMPAcT* that allows extracting the HFSM of an Android application by a black-box dynamic exploration approach.

At the conclusion of the dissertation, it is possible to verify that FSM generated after the test of the Call Pattern is the expected one and that through the *REiMPAcT* approach it is possible to analyze with more detail the Android application under test through the generated HFSM and the collected metrics.

The *iMPAcT Tool* offers significant value to the field of Android applications testing, but it still has great potential to grow in terms of the Patterns Catalog and types of actions that can be automatically identified by the dynamic exploration of the tool. The work achieved with the extension of the tool with reverse engineering capabilities (*REiMPAcT* approach) can also be improved and extended. The visualization of HFSM can be improved through its exportation to PDF. All the changes performed on the *iMPAcT Tool* aim to ensure the quality and proper functioning of the Android applications under test and analysis.

The *REiMPAcT* approach implemented in this dissertation and, its results were translated in an article submitted to the tools and artifact (TAR) papers of the conference ISSRE 2019 [Sof].



# Bibliography

- [Nil09] Erik G. Nilsson. *Design patterns for user interface for mobile applications*. Adv. Eng. Softw, December 2009. ISBN: 40(12):1318–1328.
- [Bör95] Egon Börger, ed. *Specification and Validation Methods*. New York, NY, USA: Oxford University Press, Inc., 1995. ISBN: 0-19-853854-5.
- [LY96] D. Lee and M. Yannakakis. “Principles and methods of testing finite state machines—a survey”. In: *Proceedings of the IEEE* 84.8 (Aug. 1996), pp. 1090–1123. ISSN: 0018-9219. DOI: [10.1109/5.533956](https://doi.org/10.1109/5.533956).
- [Ris98] Linda Rising. *The Patterns Handbook: Techniques, Strategies and Applications*. Fourth. Cambridge University Press, 1998.
- [Ral02] Richard Helm & John Vlissides. Ralph Johnson Erich Gamma. *Design Patterns – Elements of Reusable Object-Oriented Software*. 2002.
- [MBN03] A. Memon, I. Banerjee, and A. Nagarajan. “GUI ripping: reverse engineering of graphical user interfaces for testing”. In: *10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings*. Nov. 2003, pp. 260–269. DOI: [10.1109/WCRE.2003.1287256](https://doi.org/10.1109/WCRE.2003.1287256).
- [DB05] Arie van Deursen and Elizabeth Burd. “Software reverse engineering”. In: *Journal of Systems and Software* 77 (Sept. 2005), pp. 209–211. DOI: [10.1016/j.jss.2004.03.031](https://doi.org/10.1016/j.jss.2004.03.031).
- [Nik+05] Ana Paiva Nikolai et al. “Modeling and Testing Hierarchical GUIs”. In: *Proc.ASM05. Université de Paris 12*. 2005, pp. 8–11.
- [UL07] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007. ISBN: 0123725011, 9780080466484.
- [RHR08] A. Rohatgi, A. Hamou-Lhadj, and J. Rilling. “An Approach for Mapping Features to Code Based on Static and Dynamic Analysis”. In: *2008 16th IEEE International Conference on Program Comprehension*. June 2008, pp. 236–241. DOI: [10.1109/ICPC.2008.35](https://doi.org/10.1109/ICPC.2008.35).

- [AFT09] D. Amalfitano, A. R. Fasolino, and P. Tramontana. “Experimenting a reverse engineering technique for modelling the behaviour of rich internet applications”. In: *2009 IEEE International Conference on Software Maintenance*. Sept. 2009, pp. 571–574. DOI: [10.1109/ICSM.2009.5306391](https://doi.org/10.1109/ICSM.2009.5306391).
- [GPF10] A. M. P. Grilo, A. C. R. Paiva, and J. P. Faria. “Reverse engineering of GUI models for testing”. In: *5th Iberian Conference on Information Systems and Technologies*. June 2010, pp. 1–6.
- [MTR10] A. Marchetto, P. Tonella, and F. Ricca. “Under and Over Approximation of State Models Recovered for Ajax Applications”. In: *2010 14th European Conference on Software Maintenance and Reengineering*. Mar. 2010, pp. 236–239. DOI: [10.1109/CSMR.2010.42](https://doi.org/10.1109/CSMR.2010.42).
- [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. Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 77–83. ISBN: 978-1-4503-0592-1. DOI: [10.1145/1982595.1982612](https://doi.org/10.1145/1982595.1982612). URL: <http://doi.acm.org/10.1145/1982595.1982612>.
- [CPP12] Inês Coimbra Morgado, Ana C. R. Paiva, and João Pascoal Faria. “Dynamic Reverse Engineering of Graphical User Interfaces”. In: *International Journal On Advances in Software* 5.3 and 4 (2012), pp. 224–236. ISSN: 1095-1350. DOI: [10.1109/WCRE.2007.44](https://doi.org/10.1109/WCRE.2007.44). URL: <http://goo.gl/yRoIKF>.
- [MDL12] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. “Crawling Ajax-Based Web Applications Through Dynamic Analysis of User Interface State Changes”. In: *ACM Trans. Web* 6.1 (Mar. 2012), 3:1–3:30. ISSN: 1559-1131. DOI: [10.1145/2109205.2109208](https://doi.org/10.1145/2109205.2109208). URL: <http://doi.acm.org/10.1145/2109205.2109208>.
- [MP13] T. Monteiro and A. C. R. Paiva. “Pattern Based GUI Testing Modeling Environment”. In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. Mar. 2013, pp. 140–143. DOI: [10.1109/ICSTW.2013.24](https://doi.org/10.1109/ICSTW.2013.24).
- [YPX13] Wei Yang, Mukul R. Prasad, and Tao Xie. “A Grey-box Approach for Automated GUI-model Generation of Mobile Applications”. In: *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering*. FASE’13. Rome, Italy: Springer-Verlag, 2013, pp. 250–265. ISBN: 978-3-642-37056-4. DOI: [10.1007/978-3-642-37057-1\\_19](https://doi.org/10.1007/978-3-642-37057-1_19). URL: [http://dx.doi.org/10.1007/978-3-642-37057-1\\_19](http://dx.doi.org/10.1007/978-3-642-37057-1_19).
- [MP14a] R. M. L. M. Moreira and A. C. R. Paiva. “A GUI modeling DSL for pattern-based GUI testing PARADIGM”. In: *2014 9th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*. Apr. 2014, pp. 1–10.

- [MP14b] Rodrigo M.L.M. Moreira and Ana C.R. Paiva. “PBGT Tool: An Integrated Modeling and Testing Environment for Pattern-based GUI Testing”. In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ASE ’14. Vasteras, Sweden: ACM, 2014, pp. 863–866. ISBN: 978-1-4503-3013-8. DOI: [10.1145/2642937.2648618](https://doi.org/10.1145/2642937.2648618). URL: <http://doi.acm.org/10.1145/2642937.2648618>.
- [MPF14] I. C. Morgado, A. C. R. Paiva, and J. P. Faria. “Automated Pattern-Based Testing of Mobile Applications”. In: *2014 9th International Conference on the Quality of Information and Communications Technology*. Sept. 2014, pp. 294–299. DOI: [10.1109/QUATIC.2014.47](https://doi.org/10.1109/QUATIC.2014.47).
- [SP14] C. Sacramento and A. C. R. Paiva. “Web Application Model Generation through Reverse Engineering and UI Pattern Inferring”. In: *2014 9th International Conference on the Quality of Information and Communications Technology*. Sept. 2014, pp. 105–115. DOI: [10.1109/QUATIC.2014.20](https://doi.org/10.1109/QUATIC.2014.20).
- [MP15a] I. C. Morgado and A. C. R. Paiva. “Testing Approach for Mobile Applications through Reverse Engineering of UI Patterns”. In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*. Nov. 2015, pp. 42–49. DOI: [10.1109/ASEW.2015.11](https://doi.org/10.1109/ASEW.2015.11).
- [MP15b] I. C. Morgado and A. C. R. Paiva. “The iMPAcT Tool: Testing UI Patterns on Mobile Applications”. In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Nov. 2015, pp. 876–881. DOI: [10.1109/ASE.2015.96](https://doi.org/10.1109/ASE.2015.96).
- [MP15c] Inês Coimbra Morgado and Ana C. R. Paiva. “Test Patterns for Android Mobile Applications”. In: *Proceedings of the 20th European Conference on Pattern Languages of Programs*. EuroPLoP ’15. Kaufbeuren, Germany: ACM, 2015, 32:1–32:7. ISBN: 978-1-4503-3847-9. DOI: [10.1145/2855321.2855354](https://doi.org/10.1145/2855321.2855354). URL: <http://doi.acm.org/10.1145/2855321.2855354>.
- [CP17] Inês Coimbra Morgado and Ana Paiva. “Mobile GUI testing”. In: *Software Quality Journal* 26 (Sept. 2017). DOI: [10.1007/s11219-017-9387-1](https://doi.org/10.1007/s11219-017-9387-1).
- [CP18] Inês Coimbra Morgado and Ana Paiva. “The iMPAcT Tool for Android Testing”. In: *Proceedings of the ACM on Human-Computer Interaction* 3 (June 2018), pp. 1–23. DOI: [10.1145/3300963](https://doi.org/10.1145/3300963).
- [MP19] Inês Coimbra Morgado and Ana C. R. Paiva. “The iMPAcT Tool for Android Testing”. In: *Proc. ACM Hum.-Comput. Interact.* 3.EICS (June 2019), 4:1–4:23. ISSN: 2573-0142. DOI: [10.1145/3300963](https://doi.org/10.1145/3300963). URL: <http://doi.acm.org/10.1145/3300963>.
- [And] Material Design for Android. URL: <https://developer.android.com/guide/topics/ui/look-and-feel> (visited on 06/18/2019).

- [BA] Nick Butcher and Android Developers. 2016. Android Design in Action: Navigation Anti-Patterns. URL: <http://www.allreadable.com/ff64728N> (visited on 12/13/2018).
- [Cal] Calabash Framework. URL: <https://calaba.sh/> (visited on 04/04/2019).
- [Cuc] Cucumber. URL: <https://cucumber.io/> (visited on 04/04/2019).
- [Des] Android Design. URL: <https://developer.android.com/design/> (visited on 12/12/2018).
- [Deva] Android Developers. URL: <https://developer.android.com/index.html> (visited on 06/18/2019).
- [Devb] Android Developers. *Android Development Guide*. URL: <https://developer.android.com/guide/> (visited on 12/14/2018).
- [Dra] Side Drawer. URL: <https://material.io/design/components/navigation-drawer.html> (visited on 12/12/2018).
- [Fraa] Appium Framework. URL: <http://appium.io/> (visited on 11/29/2018).
- [Frab] CRAWLJAX Framework. URL: <http://crawljax.com/> (visited on 12/04/2018).
- [Frac] Eclipse Modeling Framework. URL: <https://www.eclipse.org/modeling/emf/> (visited on 12/03/2018).
- [Frad] Espresso Framework. Available at <https://developer.android.com/training/testing/espresso/>. (Visited on 11/29/2018).
- [Frae] Robotium Framework. URL: <http://www.robotium.org> (visited on 11/29/2018).
- [Fraf] UI Automator Framework. URL: <https://developer.android.com/training/testing/ui-automator> (visited on 11/29/2018).
- [Gui] Android Quality Guidelines. URL: <https://developer.android.com/docs/quality-guidelines/core-app-quality> (visited on 12/12/2018).
- [GUI] GUITAR Framework. URL: <https://sourceforge.net/projects/guitar/> (visited on 12/04/2018).
- [Mon] Monkey. URL: <https://developer.android.com/studio/test/monkey> (visited on 12/04/2018).
- [Nag] Back Navigation. URL: <https://developer.android.com/training/implementing-navigation/temporal> (visited on 12/14/2018).
- [Nav] Up Navigation. URL: <https://developer.android.com/training/implementing-navigation/ancestral> (visited on 12/14/2018).
- [Qua] Core App Quality. URL: <https://developer.android.com/docs/quality-guidelines/core-app-quality> (visited on 06/18/2019).
- [Sof] The 30th International Symposium on Software Reliability Engineering (ISSRE 2019). URL: <http://2019.issre.net/> (visited on 06/24/2019).



- [Staa] Statista. *Focus areas for testing mobile applications from 2013 to 2017*. URL: <https://www.statista.com/statistics/500605/worldwide-mobile-application-testing-focus-areas/> (visited on 11/27/2018).
- [Stab] Statista. *Global mobile OS market share 2009-2018*. URL: <https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/> (visited on 11/24/2018).
- [Stac] Statista. *Number of available applications in the Google Play Store from December 2009 to September 2018*. URL: <https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/> (visited on 11/27/2018).
- [Stad] Statista. *Number of mobile app downloads worldwide in 2017, 2018 and 2022*. URL: <https://www.statista.com/statistics/271644/worldwide-free-and-paid-mobile-app-store-downloads/> (visited on 11/19/2018).
- [Stae] Statista. *Number of smartphone users worldwide from 2014 to 2020*. URL: <https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/> (visited on 11/19/2018).
- [Staf] Statista. *Smartphone user penetration as percentage of total global population from 2014 to 2021*. URL: <https://www.statista.com/statistics/203734/global-smartphone-penetration-per-capita-since-2005/> (visited on 11/27/2018).
- [Tab] Tabs. URL: <https://material.io/design/components/tabs.html> (visited on 12/13/2018).
- [TS] Understand Tasks and Back Stack. URL: <https://developer.android.com/guide/components/activities/tasks-and-back-stack> (visited on 12/13/2018).
- [Tes] Android Testing. URL: <https://developer.android.com/training/testing/> (visited on 12/12/2018).



# Appendix A

## Results of the Experiments

| <b>Application</b> | <b>Call Pattern FSM<br/>(5min)</b> | <b>Number of activities<br/>extracted dynamically (5min)</b> |
|--------------------|------------------------------------|--|
| aMetro             | Yes                                | 3  |
| Paris Metro        | Yes                                | 5  |
| EasyBus Porto      | Yes                                | 5  |
| Forest Fires       | Yes                                | 4  |
| Portugal Newspaper | Yes                                | 6  |
| Sapo Newspaper     | Yes                                | 4  |
| Google News        | Yes                                | 5  |
| Math tricks        | Yes                                | 6  |
| Pilot              | Yes                                | 6  |
| Periodic Table     | Yes                                | 4  |
| MyResults          | Yes                                | 5  |
| Hockey             | Yes                                | 3  |
| Monefy             | Yes                                | 4  |
| 1Money             | Yes                                | 5  |
| Home Workout       | Yes                                | 5  |
| Pedometer          | Yes                                | 5  |
| Wysa               | Yes                                | 4  |
| Moodpath           | Yes                                | 3  |
| Wikipedia          | Yes                                | 6  |
| Poems              | Yes                                | 5  |

Table A.1: Call Test Pattern results (FSM construction results) and number of activities explored dynamically by the *iMPAcT Tool* (with the *REiMPAcT* component).

| <b>Application</b> | <b>Exploration %<br/>(5min)</b> | <b>Exploration %<br/>(15min)</b> |
|--------------------|---------------------------------|----------------------------------|
| aMetro             | 50                              | 67                               |
| Paris Metro        | 13                              | 18                               |
| EasyBus Porto      | 56                              | 67                               |
| Forest Fires       | 67                              | 83                               |
| Portugal Newspaper | 40                              | 40                               |
| Sapo Newspaper     | 22                              | 28                               |
| Google News        | 11                              | 18                               |
| Math Tricks        | 18                              | 30                               |
| Pilot              | 55                              | 55                               |
| Periodic Table     | 31                              | 38                               |
| MyResults          | 17                              | 33                               |
| Hockey             | 43                              | 57                               |
| Monefy             | 19                              | 29                               |
| 1Money             | 29                              | 35                               |
| Home Workout       | 26                              | 37                               |
| Pedometer          | 11                              | 17                               |
| Wysa               | 15                              | 23                               |
| Moodpath           | 13                              | 17                               |
| Wikipedia          | 16                              | 22                               |
| Poems              | 45                              | 64                               |
| <b>Average</b>     | <b>30</b>                       | <b>39</b>                        |

Table A.2: Percentage of activities explored by the *iMPAcT Tool* (with the *REiMPAcT* component).