



FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Automated Pattern-Based Testing of Mobile Applications

Inês Coimbra Morgado

Supervisor: Ana C. R. Paiva

Co-Supervisor: João Pascoal Faria

Programa Doutoral em Engenharia Informática

January 11, 2017

Faculdade de Engenharia da Universidade do Porto

Automated Pattern-Based Testing of Mobile Applications

Inês Coimbra Morgado

Dissertation submitted to Faculdade de Engenharia da Universidade do Porto
to obtain the degree of

Doctor Philosophiae in Informatics Engineering

President: Eugénio da Costa Oliveira

Referee: Alberto Manuel Rodrigues da Silva

Referee: José Creissac Campos

Referee: Miguel António Sousa Abrunhosa Brito

Referee: Raul Fernando de Almeida Moreira Vidal

Supervisor: Ana Cristina Ramada Paiva

January 11, 2017

Abstract

Testing is essential to improve the quality of a product and is, thus, a crucial part of any development process. However, the amount of time and resources companies dedicate to it is usually reduced, which justifies the high focus of the community on improving testing automation. Considering the importance smartphones have acquired in our daily lives, it is extremely important to define techniques to test mobile applications in order to ensure their quality.

This document presents a testing approach and tool (iMPAcT) to improve the automation of mobile testing based on the presence of recurring behaviour, UI Patterns. It combines reverse engineering, pattern matching and testing. The reverse engineering process is responsible for crawling the application, *i.e.* to analyse the state of the application and to interact with it by firing events. The pattern matching tries to identify the presence of UI patterns based on a catalogue of patterns. When a UI Pattern from the catalogue is detected, a test strategy is applied. These test strategies are called UI Test Patterns. These three phases work in an iterative way: the patterns are identified and tested between firing of events, *i.e.* the process alternates between exploring the application and testing the UI Patterns. The process is dynamic and fully automatic not requiring any previous knowledge about the application under test.

A catalogue of patterns is presented in this document. It has UI Patterns to identify on mobile applications and UI Test Patterns defining generic test strategies to test them. These patterns are generic and, thus, can be applied to any application.

In order to validate the overall approach we have conducted three experiments. The goal is to analyse the capacity of iMPAcT of finding failures and assess if the results are reliable. In addition, we also measured the execution time and events coverage.

Keywords: Mobile. Testing. Reverse Engineering. Android. UI Patterns. Test Patterns. GUI Testing.

Resumo

O teste é essencial para melhorar a qualidade de um produto sendo, por conseguinte, uma parte crucial do processo de desenvolvimento. No entanto, a quantidade de tempo e recursos que as empresas lhe dedicam é, geralmente, reduzido, o que justifica o foco da comunidade na melhoria da automação de testes.

Este documento apresenta uma abordagem de teste, e respetiva ferramenta (iMPAcT), para melhorar a automação de teste de aplicações móveis baseado na presença de comportamento recorrente, padrões de UI. Esta abordagem combina engenharia reversa, deteção de padrões e teste. O processo de engenharia reversa é responsável pela exploração da aplicação, ou seja, pela análise do estado da aplicação e a interação com ela através de eventos. O processo de deteção de padrões baseia-se num catálogo de padrões para identificar padrões de UI. Quando um padrão de UI do catálogo é detetado, uma estratégia de teste é aplicada. Estas estratégias de teste denominam-se padrões de teste. Estas três fases trabalham de forma iterativa: os padrões são identificados e testados entre eventos, ou seja, o processo alterna entre a exploração da aplicação e o teste dos padrões de UI. O processo é dinâmico e completamente automático e não necessita de qualquer conhecimento prévio acerca da aplicação a testar.

Um catálogo de padrões é apresentado neste documento. Contém padrões de UI para serem identificados em aplicações móveis e padrões de teste que definem estratégias de teste genéricas para os testar. Estes padrões são genéricos, podendo ser aplicados a qualquer aplicação.

Para validar a abordagem, foram realizados três experiências. O objetivo é analisar a capacidade da ferramenta (iMPAcT) de detetar falhas e verificar se os resultados são fiáveis. Para além disso, também se mediu o tempo de execução e a cobertura de eventos.

Keywords: Teste de aplicações móveis. Android. Padrões de UI. Padrões de teste. Teste de GUI.

Acknowledgments

First of all I would like to thank my supervisor, Ana Paiva, for her guidance and support during the development of this work. The moments of discussion were essential to its success. Moreover, her encouragement and motivation were extremely valuable.

I would also like to thank my co-supervisor, João Pascoal Faria, for his valuable inputs throughout the work, specially during the definition of the problem.

My thanks to professors Atif Memon and José Creissac Campos for their input at the thesis proposal meeting.

I wish to express my gratitude to Luis Mesquita who has been an important part on completing the final stages of this work and to professor Cristina for reviewing the English of this document.

I thank my friends Alice, André, Daniela, Pedro and Susana, whose friendship is invaluable to me. A special thanks to Susana who is also taking her PhD and, thus, with whom I could discuss the different stages of taking a PhD. Your insight was extremely helpful.

My sisters, Sara, Bia and Sofia, are also a huge part of my life and I would not be the same person without them. Thank you all for putting up with me and for teasing me. A special thanks to Sara whose support has been inestimable.

I would also like to thank my aunt, Olga, for her friendship throughout all these years, and to my parents, Ângela and Paulo, who have taught me the most important values with which I rule my life and who have been true friends to me. A special thanks to my mother for all her caring and for all the love and concern she dedicates to me.

Finally, thank you Rui. Thank you for your love, for your understanding and for your friendship. Thank you for all the adventures we have shared and for all the amazing years we have cherished with each other. And thank you for helping me reviewing this document.

Inês Coimbra Morgado

“Não sei por onde vou, sei que não vou por aí.”

José Régio

Contents

List of Figures	xiii
List of Tables	xvi
List of Abbreviations	xvii
1 Introduction	1
1.1 The Challenge	1
1.1.1 Testing	2
1.1.2 Mobile Testing	3
1.1.3 Reverse Engineering	4
1.1.4 Patterns	6
1.2 Research Goal	7
1.3 Methodology	8
1.4 Contributions	10
1.5 Overview of the Dissertation	12
2 Mobile Testing	13
2.1 Mobile Testing Automation	14
2.2 Reverse Engineering for Mobile Testing	21
2.3 Pattern-Based Testing	29
2.4 Android Mobile Test Automation Frameworks	31
2.5 Main Conclusions	36
3 Pattern Based Testing	37
3.1 Definitions	37
3.2 General Overview	41
3.3 Catalogue of Patterns	43
3.3.1 Side Drawer Pattern	43
3.3.2 Orientation Pattern	45
3.3.3 Resource Dependency Pattern	47
3.3.4 Tab Pattern	48
3.4 Exploration	49
3.5 Pattern Matching	53
3.6 Testing	53
3.7 Artefacts	53

3.7.1	Report	53
3.7.2	Model of the Observed Behaviour	56
3.8	Conclusions	57
4	Implementation	59
4.1	Operating System	59
4.2	Technological Requirements	59
4.3	Android Mobile Applications	62
4.4	Process for Defining the Heuristics	63
4.5	The iMPAcT Tool	64
4.5.1	iMPAcT's Configurator	64
4.5.2	iMPAcT's Tester	66
4.6	iMPAcT's implementation	67
4.6.1	Exploration	68
4.6.2	Pattern Matching	72
4.6.3	Testing	72
4.7	Implementation details	73
4.8	Patterns Catalogue	86
4.8.1	Side Drawer Pattern	86
4.8.2	Orientation Pattern	88
4.8.3	Resource Dependency Pattern	89
4.8.4	Tab Pattern	90
4.9	Conclusions	92
5	Experiments	95
5.1	Research Questions	95
5.2	Technical Specifications	96
5.3	Detection of UI failures	96
5.4	Impact of exploration algorithms and pattern identification order on finding failures	101
5.5	Quality of the tool results	104
5.6	Visualisation of Results	110
5.7	Conclusions	111
6	Discussion	113
6.1	RQ1: Is iMPAcT able to identify failures in Android Mobile Applications?	113
6.2	RQ2: Is iMPAcT's performance affected by the size of the applications?	114
6.3	RQ3: Does the exploration algorithm used affect the results, performance and coverage of iMPAcT?	115
6.4	RQ4: Does the order in which the patterns are identified and tested affect the results, performance and coverage of iMPAcT?	117
6.5	RQ5: Are the results provided by iMPAcT sensitive, <i>i.e.</i> are the existing failures properly identified by iMPAcT?	118
6.6	RQ6: Does iMPAcT reliably identify failures, <i>i.e.</i> are the failures detected by iMPAcT actually failures?	119
6.7	Conclusions	119

<i>CONTENTS</i>	xi
7 Conclusions	121
References	125
Index	137
A Results for the “Quality of the Results” Experiment	137

List of Figures

1.1	Architecture of a reverse engineering process	4
3.1	List corresponding layout in the TomDroid application	38
3.2	Components Diagram of the Approach	41
3.3	Action Bar of the Google Slides application	43
3.4	Example of the side drawer UI Pattern [Android, 2015b]	44
3.5	Example of rotation in the Google Play Music application	45
3.6	Example of a set of tabs	48
3.7	Example of a menu button (on the right)	51
3.8	Example of a finite state machine	56
4.1	Example of the <i>uiautomatorviewer</i> tool on selecting the App Button of the Google Calendar application	62
4.2	Main window of the iMPAcT tool's GUI	64
4.3	Window to define the order in which the patterns will be tested	65
4.4	Window to select the patterns	66
4.5	Window to configure the Resource Dependency Pattern	66
4.6	Class Diagram of the Data Structure	70
4.7	Example of a crash	76
4.8	Example of a list that has been scrolled but has already reached its end (from Android Settings)	78
4.9	The same contextual menu displayed in two different positions	79
4.10	Element "Example" is the same regardless of its position	80
4.11	Check box and corresponding layout in the QuickLyric application	82
5.1	Rotating the screen makes the pop-up disappear	99
5.2	Screen of Book Catalogue that contains the tabs: "Details" and "Notes". Horizontally swiping the screen does not change the selected tab.	99
5.3	Rotating the screen makes the Search option disappear	100
5.4	The Side Drawer of the Calendar application does not take up the full height of the screen	100
5.5	State Machine of the Tippy Tipper application	111

List of Tables

4.1	Matching of the non-official frameworks with the approach requirements .	61
4.2	Matching of the official frameworks with the approach requirements . . .	61
5.1	Applications tested with the iMPAcT tool	97
5.2	Summary of test results part 1: FF-Failure Found; AF-Absence of Fail- ures; NA-Not Applicable	98
5.3	Summary of test results part 2	101
5.4	Results obtained when applying different configurations to Tomdroid - Part 1	103
5.5	Results obtained when applying different configurations to Tomdroid - Part 2	103
5.6	Results obtained when applying the different exploration algorithms and pattern orders to Book Catalogue - Part 1	104
5.7	Results obtained when applying the different exploration algorithms and pattern orders to Book Catalogue - part 2	104
5.8	Definition of positives and negatives	105
5.9	Final applications selection	108
5.10	Final results of true/false positives/negatives for the Side Drawer Pattern .	109
5.11	Final results of true/false positives/negatives for the Orientation Pattern .	109
5.12	Final results of true/false positives/negatives for the Tab Pattern	109
5.13	Final Results	110
A.1	Results for the Side Drawer Pattern for the JW Library application	137
A.2	Results for the Orientation Pattern for the JW Library application	138
A.3	Results for the Tab Pattern for the JW Library application	138
A.4	Results for the Tab Pattern for the Scribd application	138
A.5	Results for the Side Drawer Pattern for the Bible application	139
A.6	Results for the Orientation Pattern for the Bible application	139
A.7	Results for the Tab Pattern for the Bible application	139
A.8	Results for the Side Drawer Pattern for the Google Play Books application	139
A.9	Results for the Orientation Pattern for the Google Play Books application	140
A.10	Results for the Tab Pattern for the Google Play Books application	140
A.11	Results for the Side Drawer Pattern for the Wikipedia application	140
A.12	Results for the Orientation Pattern for the Wikipedia application	140
A.13	Results for the Orientation Pattern for the Job Search application	141
A.14	Results for the Side Drawer Pattern for the File Commander application .	141

A.15 Results for the Orientation Pattern for the File Commander application . . .	141
A.16 Results for the Tab Pattern for the Call Blocker Free application	142
A.17 Results for the Orientation Pattern for the Trovit application	142
A.18 Results for the Side Drawer Pattern for the BZ Reminder application . . .	142
A.19 Results for the Orientation Pattern for the BZ Reminder application . . .	142
A.20 Results for the Side Drawer Pattern for the Marvel’s Comics application .	143
A.21 Results for the Orientation Pattern for the Marvel’s Comics application . .	143
A.22 Results for the Tab Pattern for the Marvel’s Comics application	143
A.23 Results for the Side Drawer Pattern for the Drawing Cartoons 2 application	144
A.24 Results for the Tab Pattern for the Drawing Cartoons 2 application	144
A.25 Results for the Side Drawer Pattern for the Draw Anime application . . .	144
A.26 Results for the Side Drawer Pattern for the DC Comics application	144
A.27 Results for the Orientation Pattern for the DC Comics application	145
A.28 Results for the Tab Pattern for the DC Comics application	145
A.29 Results for the Side Drawer Pattern for the Comics application	145
A.30 Results for the Orientation Pattern for the Comics application	145
A.31 Results for the Tab Pattern for the Comics application	146
A.32 Results for the Side Drawer Pattern for the Money Lover application . . .	146
A.33 Results for the Orientation Pattern for the Money Lover application . . .	146
A.34 Results for the Tab Pattern for the Money Lover application	146
A.35 Results for the Side Drawer Pattern for the Investing.com application . . .	147
A.36 Results for the Tab Pattern for the Investing.com application	147
A.37 Results for the Side Drawer Pattern for the Moneyfy application	147
A.38 Results for the Side Drawer Pattern for the MSN Money application . . .	147
A.39 Results for the Tab Pattern for the MSN Money application	148
A.40 Results for the Side Drawer Pattern for the Meta Trader 4 application . . .	148
A.41 Results for the Orientation Pattern for the Meta Trader 4 application . . .	148
A.42 Results for the Side Drawer Pattern for the Clue - Period Tracker application	149
A.43 Results for the Orientation Pattern for the Clue - Period Tracker application	149
A.44 Results for the Tab Pattern for the Clue - Period Tracker application . . .	149
A.45 Results for the Tab Pattern for the Pedometer application	149
A.46 Results for the Tab Pattern for the 7 Minute Workout application	150
A.47 Results for the Side Drawer Pattern for the ABS Workout application . . .	150
A.48 Results for the Tab Pattern for the S Health application	150

List of Abbreviations

API	Application Programming Interface
APK	Android application PacKage
AUT	Application Under Test
FSM	Finite State Machine
GOF	Gang of Four
GUI	Graphical User Interface
MBT	Model Based Testing
JPF	Java Path Finder
OP	Orientation Pattern
OS	Operating System
RE	Reverse Engineering
RQ	Research Question
SDK	Software Development Kit
SDP	Side Drawer Pattern
UI	User Interface
VM	Virtual Machine
V&V	Verification and Validation
XML	Extensible Markup Language

Chapter 1

Introduction

Smartphones have been a part of our lives for nearly a decade: the first iPhone was presented in 2007 [Apple, 2007], closely followed by Android in 2008 [Android, 2008]. In late 2013, smartphones already represented almost 60% of the mobile sales worldwide [Gartner, 2013], surpassing the one billions units sold threshold in 2014 [Gartner, 2015] with still growing numbers [Gartner, 2016a]. Moreover, in 2013 both Android's *Google Play* and Apple's *App Store* reached one million available applications and the number of downloads crossed the fifty billion threshold for Android and sixty billion for Apple [H., 2013; Ingraham, 2013].

These numbers illustrate the overwhelming importance smartphones have gained in our daily lives and how much we rely on them. This means it is of high importance to ensure the functional and security correctness of the mobile applications. One way of increasing the quality of software applications is to invest in testing and, more specifically, in test automation.

1.1 The Challenge

It is irrefutable that testing in all its forms, including software testing, is imperative in this technological world. Every product and service a company provides will face the scrutiny of the people and, thus, ensuring their quality is a requisite for any company's sustainability.

Nowadays, one of the most important forms for a company to provide its services to their clients is through mobile applications, which makes it extremely important to test them. In fact, according to World Quality Reports of 2014-15 [Capgemini et al., 2014] and 2015-16 [Capgemini et al., 2015], the number of organisations performing mobile testing has grown from 31% in 2012 to 55% in 2013, and to near 87% in 2014 and 92%

in 2015. According to these reports, the greatest challenge companies face when testing mobile applications is the lack of the right testing processes and methods, followed by insufficient time to test and the absence of in-house mobile test environments. Even though companies usually want to spend as little time and resources as possible in the testing process, the 2015-16 report states that in 2015 there was a 9% increase in the amount of Information Technology budget allocated to Quality Assurance and Testing. Therefore, it is important to invest in improving the automation of mobile testing. Moreover, a relevant part of applications present on the markets are developed by solo developers who may lack the knowledge or the time to properly test their applications and, thus, require tools to automate this process.

All these factors point to the need of automating the testing of mobile applications, which is not a trivial task. When testing mobile applications one must consider aspects that are not present neither on desktop nor on web applications, such as new development concepts like activities, new interaction gestures (*e.g.*, long-click, pinch) and limited memory [Amalfitano et al., 2012b; Muccini et al., 2012]. Moreover, mobile applications depend on the state of the mobile device: the orientation of the device, the state of the different sensors and services, the possibility of receiving a call, among other aspects. Hence, it is necessary to adapt the testing process to mobile applications instead of straightforwardly applying them and even to develop new testing methodologies.

1.1.1 Testing

When automating testing, the focus can either be on the automation of the test cases execution or on their generation. Even though the test case execution needs to tackle problems such as the variety of devices and platforms, there are already some frameworks and Application Programming Interfaces (API) focused on this, such as UIAutomator¹, Espresso² and Robotium³ for Android and an UI Automation API⁴ (Application Programming Interface) for iOS. As such, the community has been mainly focusing on automating the generation of test cases.

There are four main testing approaches for test automation of Graphical User Interfaces (GUI): random testing, unit testing, capture-replay testing and Model-Based Testing (MBT).

Random testing implies no prior knowledge of the applications and consists in randomly firing events on the application being only capable of detecting crashes. Their

¹<https://developer.android.com/training/testing/ui-testing/uiautomator-testing.html>

²<https://developer.android.com/training/testing/ui-testing/espresso-testing.html>

³<http://robotium.com/>

⁴<https://goo.gl/R2KSNE>

output is the failures (crashes) found and the execution traces. This is the only technique that does not require any manual effort. Android provides a random testing tool, Monkey⁵ that generates pseudo-random streams of user events. This tool is used by some researchers when improving test automation as it automatically provides a set of inputs that can be used by other approaches (*e.g.*, [Hu & Neamtiu, 2011]).

Unit testing is a level of testing that focuses on testing individual hardware or software units or groups of related units [IEEE, 1990], *e.g.*, if methods are considered units then there must be tests that verify the correct functioning of every method. Building the test cases is usually a manual and time consuming process requiring full knowledge about the application and its implementation. However, once the tests are implemented, they can be automatically executed with no extra effort. Unit tests can be used, for instance, for regression and integration testing and is not limited to GUI Testing.

Capture-replay testing is a technique that, as the name indicates, is divided in two phases: 1) capture the tests and 2) replay them, *i.e.* the tool records the interaction of the user with the application (capture phase) and then plays the same interaction verifying if the result is the same (replay phase). This technique is only used for GUI testing. Its main advantage is that it ensures the GUI of the application responds as it should. However, whenever there is a change on the GUI, the corresponding tests have to be recorded again (even if it is just a change in the dimensions or position of a button).

Finally, MBT [Utting & Legeard, 2006] is a technique which receives a model of the application's behaviour as input and outputs a test suite to be run on it. Its main advantage is that it generates test cases capable of thoroughly testing the application under testing (AUT). The two main issues of MBT are 1) the need for an application model, whose manual construction is a time consuming and failure prone process and 2) the combinatorial explosion of test cases. Reverse engineering techniques, which extract information from the running application or from its source code, can help tackling the first problem. To tackle the second problem either tests that are classified as redundant are removed or the tester reduces the area of focus of the tests, *i.e.* opting to only focus on some parts of the application in detriment of the whole.

1.1.2 Mobile Testing

Mobile testing, alike testing on other platforms, may focus different aspects of the application: security, usability, GUI, *etc.*. In this work, the focus is on GUI Testing because the user interface (UI) has a major impact on how the users react to the application as it is how they connect and interact with it, *i.e.* their perception of the application's behaviour

⁵<http://developer.android.com/tools/help/monkey.html>

is moulded by the information presented in the UI and how it is presented. As previously stated, techniques that have been developed for testing UIs on other platforms can not be straightforwardly applied to mobile applications, not only because of the need to adapt the techniques to the new platform but also due to the need of developing new testing techniques that are able to test aspects that are only present on mobile applications, such as the change of the device’s orientation (example in Figure 3.5 of Section 3.3.2). In fact, mobile operating systems usually provide a set of guidelines users should follow in order to improve the quality of their applications increasing the chances of it being accepted by the community [Android, 2015a; Apple, 2016; Microsoft, 2016].

1.1.3 Reverse Engineering

As previously stated, reverse engineering can help in obtaining information about the AUT to ease the application of MBT techniques.

Reverse engineering (RE) was first defined in 1985 by Rekoﬀ as

“the process of developing a set of specifications for a complex hardware system by an orderly examination of specimens of that system” [Rekoﬀ, 1985].

Five years later, Chikofsky and Cross adapted this definition to software systems:

“Reverse Engineering is the process of analysing a subject system to (1) identify the system’s components and interrelationships and (2) to create representations of the system in another form or at a higher level of abstraction” [Chikofsky & Cross, 1990].

Figure 1.1 depicts a representation of a common reverse engineering process, in which the system under analysis is analysed and abstracted in different views.

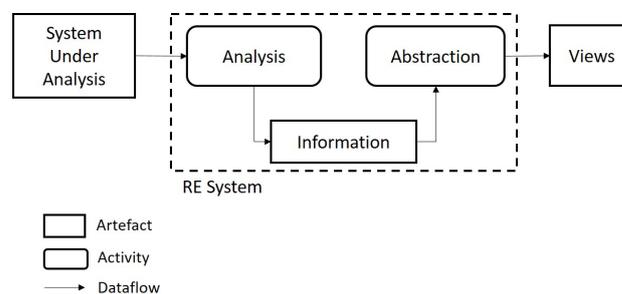


Figure 1.1: Architecture of a reverse engineering process

Even though nowadays RE is considered helpful in several areas (*e.g.*, security analysis) [Canfora & Di Penta, 2007], it initially surfaced associated with software maintenance

as it eases system comprehension. This was considered extremely important as over 50% of a system's development was occupied with maintenance tasks [Erlikh, 2000; Lientz et al., 1978; Sommerville, 1995] and over 50% of maintenance was dedicated to comprehending the system [Corbi, 1989; Standish, 1984]. RE has also proved to be useful, for instance, in coping with the Y2K problem, with the European currency conversion and with the migration of information systems to the web and towards the electronic commerce [Muller et al., 2000]. With the exploration of RE techniques, its usefulness grew from software maintenance to other fields, such as verification and validation and security analysis.

Even though RE techniques can also be used with malicious intents [Eilam, 2005], such as removal of software protection and limitations or allowing unauthorised access to systems/data, developers may also use the same techniques in order to improve the software's safety (*e.g.*, security and vulnerability auditing).

There are four types of RE: static [Binkley, 2007], in which the information is extracted from the source code or from the binary code; dynamic [Bell, 1999], in which the information is extracted from the system under execution; hybrid, a combination of both static and dynamic techniques; and historical [Kagdi et al., 2007], which obtains information on the evolution of the system kept in version control systems, such as SVN⁶ or GIT⁷. The characteristics of historical reverse engineering approaches are very different from the others and will not be considered in this work.

Static reverse engineering is performed without actually executing the code through techniques like parsing (usually returning a parse tree or an abstract syntax tree) or symbolic execution (simulating the execution according to a model, for instance). One of the advantages of using a static analysis approach is the possibility of extracting a complete sequence diagram of the program, *i.e.* how processes and events of a system operate with one another and in what order [Systems, 2012]. Obtaining this diagram can provide 100% recall [van Rijsbergen, 1979], *i.e.* all the behaviour is extracted (no false negatives), at the price of a low precision [van Rijsbergen, 1979], *i.e.* not all the extracted behaviour is real (high number of false positives) [Ng et al., 2010]. Without disregarding the importance of the information extracted by static approaches, they are unable to extract information on the system's real behaviour, *i.e.* its behaviour during runtime, and the effort of statically analysing dynamic types of object references is not conceivable for large programs [Briand et al., 2006; Ghezzi et al., 2002]. Moreover, conceptually related code is usually scattered over different source artefacts, making it a handful task to extract and analyse all the relevant information [Rothlisberger et al., 2008]. Another possible drawback is

⁶svn.apache.org

⁷git-scm.com

the necessity of the availability of the source code, which is not always feasible (when analysing grey-box components, for instance) [Mariani et al., 2011].

Dynamic analysis consists in extracting information from the program in run time, such as data-dependent execution and late binding. This provides better precision and worst recall than static analysis as the number of false positives is reduced but the number of false negatives increases [Ng et al., 2010]. Dynamic analysis is usually more complex than static analysis as it faces challenges, such as the order in which the events are identified and fired, preconditions of the application and of its running environment, when to stop the exploration (for instance, when a sufficient amount of the model is extracted) and how to represent the different states of the application: too much information will result in an explosion of the number of states whereas too little information will result in a weak representation of the application's behaviour.

One of the most popular techniques used by dynamic approaches is instrumentation, *i.e.* the ability to monitor or measure the level of a product's performance, to diagnose errors and to obtain trace information, implemented in the form of code instructions that monitor specific components in a system [Microsoft, 2013]. These approaches usually define scenarios to be executed resulting in execution traces, which can be analysed in order to extract the information relevant to the work at hands. Therefore, it is not surprising that most dynamic approaches opt to represent information as sequence diagrams, even if just as an internal and intermediate representation and, thus, not considered as an *output* artefact.

Despite the advantages of the dynamic approaches over the static ones, dynamic analyses do not provide the behaviour of the whole program [Ng et al., 2010], as they do not provide 100% recall, but only a high precision. Hybrid analysis improve the completeness, scope and precision of the analysis [Bell, 1999] as they bring together the advantages of both static and dynamic analysis. However, alike static approaches, they require access to the source or byte code. The techniques used in hybrid approaches are the combination of the ones used in both static (*e.g.*, parsing) and dynamic (*e.g.*, instrumentation) approaches as well as some that use both static and dynamic techniques like concolic execution (a mixture between concrete and symbolic execution).

In summary, each type of RE has its own advantages and disadvantages, requiring a careful analysis of the problem in order to decide which is more suitable.

1.1.4 Patterns

Patterns were initially defined in 1977 by Alexander *et al.* in the context of architecture as the

“current best guess as to what arrangement of the physical environment will work to solve the problem presented” [Alexander et al., 1977].

This concept can be generalised for all fields of studies as “an idea that has been useful in one practical context and will probably be useful in others” [Fowler, 1997] or “a named and well-known problem/solution pair that can be applied in new contexts, with advice on how to apply it in novel situations and discussion of its trade-offs, implementations, variations and so forth” [Larman, 2004].

According to the Gang of Four (GOF) book [Gamma et al., 1995], which is considered the “Bible” of design patterns [Larman, 2004], a pattern has four essential elements: *name*, which identifies the pattern and, when correctly defined, can help in the understanding of the problem and solution it represents, *problem*, which describes when the pattern can be applied, *i.e.* the situation it is solving, *solution*, which describes how to solve the problem without specifying aspects regarding the implementation, and *consequences*, which describe the outcomes, both positive and negative, of applying the pattern. Even though every pattern has all these parts, the way a pattern is described can follow different forms. The original one, the Alexandrian form, was based on the definition presented by Christopher Alexander [Alexander et al., 1977]. However, this is non-structured and lengthy and, thus, it has suffered some alterations, and different forms have surfaced, from which the most famous one is the GOF form [Gamma et al., 1995], which regardless of being an extensive form, is very structural as it breaks the pattern up into different parts (intent, motivation, applicability, structure, participants, collaborations, consequences, implementation, sample code, known uses and related patterns).

The work described in this document considers UI Patterns, *i.e.* patterns that can be found in the user interface. These patterns are present in several applications, including mobile applications. One of the most widely known UI Pattern is the login/password, as it can be found in all sorts of applications (desktop, web or mobile). Patterns specific for mobile applications can concern, for instance, the orientation of the device, the ways of navigation inside the application and the device’s sensors and services’ status reading.

1.2 Research Goal

The goal of the work presented in this document is to improve mobile testing automation without requiring any prior knowledge on the application under test. It is believed that mobile applications present recurring behaviour, UI Patterns [Neil, 2014; Nilsson, 2009; Sahami Shirazi et al., 2013], and that these can be associated with test strategies to be applied whenever they are detected. Hence, the research work described in this document

aims at testing mobile applications by providing a non-intrusive, fully automatic, iterative approach that detects and tests the UI Patterns present on mobile applications.

Moreover, a collection of some of the aforementioned recurring behaviours was gathered and is presented in this document. This collection is common to several mobile applications and can be used and improved by other approaches.

1.3 Methodology

According to Lázaro and Marcos [Lázaro & Marcos, 2005], every research work consists of, at least, four phases: topic research, *i.e.* the study of the topic involved on the problem, concept formulation, *i.e.* proposing an hypothesis that may solve the problem, approach implementation, *i.e.* the implementation of the approach that may be able to verify the hypothesis, and validation of the approach, *i.e.* verify if the approach validates the hypothesis.

Topic research

The first phase of the research consisted in collecting information on the state of the art of the problem identified. Initially, the topics involved were defined: mobile testing and mobile reverse engineering, followed by the gathering of research work mainly published in journals and conference proceedings. The different works were studied identifying its main purposes, advantages and limitations, which enabled the identification of the currently open issues. Apart from research work, the current state of the art in terms of commercial tools to automate test execution on mobile applications was also analysed. The result of this study is presented in Chapter 2.

Concept formulation

After researching and gathering information of the current state of mobile testing and its current challenges, the research hypothesis was formulated:

“Mobile applications have generic recurring behaviour, independent of their specific domain, that may be tested automatically by combining reverse engineering and testing within an iterative process.”

The automation of mobile testing can be eased by focusing the test on recurring behaviour on the application: each recurring behaviour (UI Patterns) may be associated with a test strategy (Test Pattern). RE techniques may be used to explore the AUT and extract

information on its state. This information can be studied in order to verify the presence of recurring behaviour which can then be tested. The approach to validate this hypothesis is presented in Chapter 3.

Approach implementation

The third phase consisted in implementing the approach, which entailed building a catalogue of UI Patterns and the corresponding test strategies and developing a tool to implement the approach.

Even though we believe the research hypothesis is valid for all mobile operating systems and that the approach can be applied to all of them, the implementation focused on Android mobile applications because Android is the leader of smartphone sales having gained over 80% of the smartphones market share by the end of 2015 [Gartner, 2016b].

Before starting the development of the tool, the Android Ripper tool implemented by Amalfitano *et al.* [Amalfitano et al., 2012b] was analysed in order to be used as the basis of the implementation. However, this option was abandoned as 1) the architecture of the tool did not ease its adaptation to include the identification and testing of the UI Patterns on the fly, 2) the tool can only be applied to applications running on the emulator and the intended result was for it to work on a real device and 3) it could only target applications running Android version 2.3 or lower, which was also not the intent of this work.

Hence, a new tool was developed. The first step was to structure the tool's architecture and to define one UI Pattern and the corresponding UI Test Pattern. The second step was the development of a crawler: the cyclic extraction of information and event execution. Finally, the identification and testing of the UI Patterns was implemented and integrated with the rest of the tool. Along the development, a catalogue of UI and Test Patterns was built and continuously improved and more algorithms for exploring the application and comparing elements were defined. The details of the implementation are presented in Chapter 4.

Validation

Easterbrook *et al.* [Easterbrook et al., 2008] present five different methods of validation:

- controlled experiment, the “investigation of a testable hypothesis where one or more independent variables are manipulated to measure their effect on one or more dependant variables” [Easterbrook et al., 2008];

- case study, “an empirical inquiry that investigates a contemporary phenomenon and within its real-life context, especially when the boundaries between phenomenon and context are not clearly defined” [Yin, 2002];
- survey research, “a study [such as a questionnaire] to identify the characteristics of a broad population of individuals” [Easterbrook et al., 2008];
- ethnography, “a form of research focusing on the sociology of meaning through field observation” [Easterbrook et al., 2008];
- action research, an “attempt to solve a real world problem while simultaneously studying the experience of solving the problem” [Easterbrook et al., 2008].

The research work presented in this document is validated through one case study and two experiments that analyse the usefulness, reliability and scalability of the approach. They are presented in Chapter 5 followed by the discussion of their results in Chapter 6.

1.4 Contributions

The main contributions offered by this research work are:

1. a catalogue of patterns, which indicates which UI Patterns to test and the corresponding strategies (UI Test Patterns);
2. a tool that automatically reverse engineers and tests the AUT on the fly, while also obtaining a Finite State Machine (FSM) representing the AUT.

Moreover, the tool brings some advantages to the testing community with the following characteristics [Coimbra Morgado & Paiva, 2015b]:

- **Access to Source Code:** the reverse engineering process is fully dynamic, *i.e.* the source code is never accessed and no code instrumentation is required;
- **Manual Effort:** the whole process is completely automatic and, thus, no manual effort is necessary;
- **Reuse:** UI Patterns are the representation of recurring behaviours and, thus, are present in several applications. Therefore, the patterns defined may be reused without any modification;
- **Maintenance and Evolution:** Patterns can easily be added to the catalogue and the tool’s source code structure eases the process of adding new interactions;

- **Usability:** The tool is of simple usage and it does not require any knowledge of the AUT nor of the patterns;
- **Innovation:** There is no other tool that eases the process of testing mobile applications by mixing RE techniques and UI Patterns.

In this approach the manual effort is solely associated with the assembly of the patterns catalogue, which may be reused for any Android mobile application. Hence, it has a better cost-benefit ratio than MBT approaches and it enables the detection of more (and different) failures than monkey testing tools.

These contributions were described along several conference and journal publications:

- Inês Coimbra Morgado, Ana C. R. Paiva, João Pascoal Faria, Rui Camacho, *GUI Reverse Engineering with Machine Learning*, In Realizing Artificial Intelligence Synergies in Software Engineering (RAISE), 2012 First International Workshop on, Zurich, 2012, pp. 27-31, DOI: 10.1109/RAISE.2012.6227966 (indexed in IEEE Explore and DBLP);
- Inês Coimbra Morgado, Ana C. R. Paiva, João Pascoal Faria, *Dynamic Reverse Engineering of Graphical User Interfaces*, In the International Journal On Advances in Software, vol 5, issue 3-4, pp 224-236, 2012 (indexed in ThinkMind);
- Inês Coimbra Morgado, Ana C. R. Paiva, João Pascoal Faria, *Automated Pattern-Based Testing of Mobile Applications*, In Quality of Information and Communications Technology (QUATIC), 2014 9th International Conference on the, Guimarães, 2014, pp. 294-299, DOI: 10.1109/QUATIC.2014.47 (indexed in ISI Web of Science, SCOPUS, ACM Portal, DBLP and DOI System);
- Inês Coimbra Morgado, Ana C. R. Paiva, *Test Patterns for Android Mobile Applications*, In Proceedings of the 20th European Conference on Pattern Languages of Programs (EuroPLoP '15). ACM, New York, NY, USA, Article 32, pp. 32:1-32:7, DOI: <http://dx.doi.org/10.1145/2855321.2855354> (indexed in ACM Digital Library and DBLP);
- Inês Coimbra Morgado, Ana C. R. Paiva, *The iMPAcT Tool: Testing UI Patterns on Mobile Applications*, In Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on, Lincoln, NE, 2015, pp. 876-881, DOI: 10.1109/ASE.2015.96 (indexed in IEEE Explore, ACM Digital Library and DBLP);
- Inês Coimbra Morgado, Ana 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), Lincoln, NE, 2015, pp. 42-49, DOI: 10.1109/ASEW.2015.11 (indexed in ACM digital library and DBLP);

- Inês Coimbra Morgado, Ana C. R. Paiva, *Impact of execution modes on finding Android failures*, In Procedia Computer Science, Volume 83, 2016, pp 284-291, ISSN 1877-0509, DOI: <http://dx.doi.org/10.1016/j.procs.2016.04.127> (indexed in Scopus, Thomson Reuters' Conference Proceeding Citation Index, Engineering Village and DBLP);
- Inês Coimbra Morgado, Ana C. R. Paiva, *The iMPAcT Tool for Android Testing* [submitted to International Journal of Software Engineering and Knowledge Engineering];
- Amalfitano Domenico, Inês Coimbra Morgado, Anna Rita Fasolino, Ana C. R. Paiva, Vincenzo Riccio, *Exploring GUI Failures in Mobile Applications: the consequences of the orientation change* [submitted to Journal of Software: Evolution and Process].
- Inês Coimbra Morgado, Ana C. R. Paiva, *Mobile GUI Testing* [to be submitted to a journal]

1.5 Overview of the Dissertation

The remaining of this document is structured as follows. Chapter 2 presents the result of the topic research focusing on mobile testing and mobile RE. Chapter 3 describes the approach followed to verify the research hypothesis. Chapter 4 presents approach implementation (the iMPAcT tool). Chapter 5 describes the different experiments performed to validate the approach. Chapter 6 discusses the results obtained in the experiments. Finally, Chapter 7 draws the conclusions of this work and presents future work.

Chapter 2

Mobile Testing

This Chapter presents work related to mobile testing. Section 2.1 describes mobile testing, including its main purposes and recent works on the field. Section 2.2 introduces reverse engineering and presents recent works on mobile reverse engineering. Section 2.3 describes the patterns used by some of the testing approaches to ease the testing process. Section 2.4 studies the different Android mobile test automation framework already existing on the market that may help on implementing the solution. Finally, Section 2.5 draws some conclusions on the state of the art and identifies some of its limitations.

The following terms may be useful to better understand the next sections:

Adaptive Random Testing [Chen et al., 2005]: An adaptation of random testing in which it is assumed that test cases should be as evenly spread over the entire input domain as possible;

Combinatorial Testing [Kuhn et al., 2010]: a type of dynamic testing based on the premise that many errors in software can only arise from the interaction of two or more parameters. It is useful to reduce the number of necessary tests and acts on finding the errors that are only raised from rare input combinations;

Concolic execution [Sen et al., 2005]: a mixture between CONCrete execution and symbOLIC execution, by providing concrete values to the symbolic execution making it a feasible technique;

Fuzz Testing [Takanen et al., 2008]: feeding malformed or unexpected input data to a program with the objective of revealing vulnerabilities;

Hungarian algorithm [Kuhn, 1955]: a combinatorial optimization algorithm that solves the assignment problem in polynomial time;

Machine Learning [Carbonell et al., 1983]: definition of algorithms that can learn from and make predictions on data. These algorithms are based on a model built from a training set of input observations;

Unit Testing [IEEE, 1990]: testing of individual hardware or software units or groups of related units;

2.1 Mobile Testing Automation

Considering the rapid increase of mobile applications on the markets it is of high importance to improve mobile testing techniques, more specifically techniques to improve the automation of the test process. Software testing has long been of interest both to the academia and to the industry due to the necessity of improving the quality of software. However, it is necessary to study how to adapt the testing approaches to mobile or even to define new approaches as Muccini *et al.* concluded in their study on the challenges and future research directions of mobile testing in 2012 [Muccini et al., 2012]. In this study they verified that mobile applications present new challenges in contrast to other types of applications due to the peculiarities of the mobile world, such as new interaction gestures, limited resources (such as memory), new development concepts (such as activities), context awareness, *i.e.* mobile applications often rely on data obtained by the device's sensors and connectivity, and the diversity of devices and their characteristics. Moreover, Muccini *et al.* identified the main types of testing that should be performed on mobile applications: performance and reliability testing, *i.e.* validating how the application responds to the quality and availability of both connectivity and contextual information; memory and energy testing, *i.e.* assessing the (non-)existence of memory leaks; security testing, *i.e.* controlling the access to private information, which is specially important in applications that deal with delicate information, such as e-banking applications, but also important in regular applications; GUI testing, *i.e.* assessing the correct behaviour of the application's GUI; and product line testing, *i.e.* validating the application in different devices. In 2014, Gao *et al.* [Gao et al., 2014] studied the trends on mobile testing, identifying mainly the same testing purposes as Muccini *et al.*. Nevertheless, their study showed additional testing trends: usability and internationalisation testing, *i.e.* validating the user operation flows and scenarios as well as the content of the UI; compatibility and connectivity testing, *i.e.* verifying the compatibility of hybrid applications with different browsers and different wireless connectivities; and multi-tenancy testing, *i.e.* validating the behaviour of the system. However, it is not mandatory for testers to focus on all these types of testing as they should adapt to the application's requirements and to their own testing purposes.

Moreover, Gal *et al.* identified the main mobile testing infrastructures: emulation, device, cloud and crowd-based. Emulation-based testing consists on creating a virtual

machine that recreates a mobile device, an emulator, and running the tests on it. Even though this technique is inexpensive as it only requires access to a personal computer, it is not efficient to evaluate the application in terms of quality of service and emulators often do not provide the full range of gestures. Device-based testing consist in running the tests on a real device which overcomes the limitations of the emulator but implies extra costs in acquiring the devices. Cloud testing relies on third parties devices and/or emulators which are accessible through the cloud on which the application can be installed and the tests run. When the developer wants to ensure the correct functioning of the application in a large amount of devices this is the most cost-effective option. Finally, crowd-based testing relies on a community of end users using the application. Even though this approach offers the benefits of having a community using the application in different devices without the extra cost of acquiring them, there is an uncertain validation schedule and there is no way of ensuring the quality and extension of the tests.

According to the System and Software Quality Requirements and Evaluation ISO [ISO/IEC, 2011], there are eight main purposes for Verification and Validation (V&V): functional sustainability, *i.e.* “the degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions”, performance efficiency, *i.e.* “performance relative to the amount of resources used under stated conditions”, compatibility, *i.e.* “degree to which a product, system or component can exchange information with other products, systems or components, and/or perform its required functions, while sharing the same hardware or software environment”, usability, *i.e.* “degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use”, reliability, *i.e.* “degree to which a system, product or component performs specified functions under specified conditions for a specified period of time”, security, *i.e.* “degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization”, maintainability, *i.e.* “degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers”, and portability, *i.e.* “degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operational or usage environment to another”.

The remaining of this Section describes some recent (from 2010 to now) works on mobile testing.

Functional Sustainability

Functional sustainability includes verifying if the set of functions cover all the specified tasks and user objectives (functional completeness), if they provide the correct results with the necessary degree of precision (functional correctness), and if they ease the accomplishment of the specified tasks and objectives (functional appropriateness).

Nguyen *et al.* [Nguyen et al., 2012] present an approach that combines MBT and combinatorial testing [Kuhn et al., 2010] in order to produce test cases from a model of an Android mobile application. Their approach receives the FSM of the AUT as input and its output is executable test cases and it is divided in four main phases. The first phase consists in traversing the FSM in order to generate abstract test paths. The next phase transforms the generated paths into classification trees, in which each of the second level nodes represents the sequence of events of the corresponding path and the descendant nodes represent the parameters of the event. Restrictions to the parameters are manually introduced by the tester. The third step is responsible for the test combination generation, *i.e.* a t-way combinatorial strategy is applied on the classification trees generating the possible test combinations. In order to avoid the explosion of the number of test cases, paths with repeated combinations (repeated sequences of events and inputs) are removed in a post-optimisation step. Finally, the test combinations are transformed into executable test cases in the target language. In this approach, Nguyen *et al.* do not concern themselves with how the FSM is obtained, stating that it could be built manually or using model inferring techniques. Thus, they assume the tester has a valid and complete model of the application.

Van der Merwe *et al.* [van der Merwe et al., 2012] present a model checking approach to verify Android applications as an extension to Java PathFinder¹ (JPF). This extension enables the Android AUT's code to be run on the Java Virtual Machine using different event sequences to detect when errors occur. As common Java errors are already handled by the JPF, Van der Merwe *et al.* only handle the ones specific to Android. They require as input the .class files of the AUT and the configuration file containing all the properties to be verified and the corresponding rules. Dealing with the size and complexity of this file is their main limitation.

Costa *et al.* [Costa et al., 2014] present a MBT approach to test mobile applications. Their purpose is to apply their previous approach (the Pattern-Based GUI Testing project's approach [Moreira & Paiva, 2014b]) to mobile applications, more specifically Android, in order to assess if the same approach could also be applied to mobile applications. This approach consists in building a test model of the application based on the UI patterns

¹<http://babelfish.arc.nasa.gov/trac/jpf>

present on the AUT, *i.e.* instead of modelling all the behaviour of the application, only the different UI Patterns present on the AUT and the relations between them need to be modelled. Following, they automatically generate test cases from the model [Nabuco & Paiva, 2014] and automatically apply the tests on the AUT. This model can be obtained from the documentation, it can be manually built or reverse engineering techniques may be used to ease its construction [Sacramento & Paiva, 2014].

Gorla *et al.* [Gorla et al., 2014] present an approach to verify whether the AUT's actual behaviour matched its advertised behaviour. They state this is important because a behaviour may either be malicious or expected depending on the application's purpose (*e.g.*, a tracking application is supposed to request the user's location, and a gaming application is supposed to send costly messages to a server when the user is paying for a feature). Furthermore, they report that current (2014) approaches did not take this into consideration on their reports. Thus, Gorla *et al.* present an approach with the following steps: 1) they obtain a large set of applications from the Play Store; 2) they apply a well known information retrieval technique, Latent Dirichlet Allocation [Blei et al., 2003], on the AUT's description in order to identify the main topics for each application (*e.g.*, weather, map, download); 3) they cluster applications by related topics (*e.g.*, application with the topics *travel* and *navigation* are grouped on the same cluster); 4) for each application, they identify the API calls related to the permissions it requested; and 5) they use a machine learning technique, unsupervised One-Class support vector machine anomaly classification [Amer et al., 2013], to identify the outliers for each API usage and ranks the applications within their cluster where the top applications are most likely to present anomalous usage of an API.

Reliability

Reliability indicates if the system meets the requirements under normal circumstances (maturity), if the system is operational when needed (availability), if the system operates correctly even when the hardware or the software present faults (fault tolerance) and if the system is capable of restoring the previous state when a failure occurs (recoverability).

Zhifang *et al.* [Liu et al., 2010] present an approach to improve the random generation of test cases. They focus on random testing because it is a fully automatic and non-intrusive technique. However, random testing can take long to detect bugs. Thus, they adapt the Adaptive Random Testing algorithm [Chen et al., 2005] to mobile applications by defining test case distance on the context of mobile applications. They define a test case as a sequence of events in which each event has a type and a value (*e.g.*, turning the Bluetooth on would be represented as $\langle \text{Bluetooth}, \text{true} \rangle$). In order to calculate the

distance between test cases they consider the distance between the sequences of different event types without considering the values, and the distance between sequences of the same type of events considering the values.

Franke *et al.* [Franke et al., 2012a] present an approach to test how mobile applications handle the life cycles of their activities. Even though they use unit testing [IEEE, 1990], they do not consider an unit as being the smallest part of an application as it is not possible to test the life cycle of methods or interfaces. Thus, they consider an unit as a component with a single life cycle, *i.e.* activities. They focus their tests on detecting if there is any sort of data loss when the state of application changes by applying the tests to the life cycle callback-methods. In the final callback-methods (*e.g.*, onPause() and onDestroy()) they store the information and on the initial callback-methods (*e.g.*, onStart() and onResume()) they assert that the information is the same. The decision on what to assert depends on the specification of the application. In summary, they derive the necessary assertions from the specification, then they inject them into the callback-methods and finally, they perform the action that will trigger the test. They mainly focus on testing data persistence, connection status and hardware status. This approach is implemented in a tool, AndroLift [Franke et al., 2012b].

More recently (2014), Shahriar *et al.* [Shahriar et al., 2014] present an approach to fuzz test [Takanen et al., 2008] applications to verify if they suffered from memory leaks that could cause the application to crash. They achieve this by defining a set of memory leak patterns in Android and by fuzz testing the AUT: emulate the memory leak cause and verify the result. In order to define the memory leak patterns they manually analyse the errors related to memory leak issues reported on Stack Overflow², identifying the common sources of the leakages. They detect potential leaks by applying fuzz testing to emulate the identified memory leak pattern on the application. They perform three types of fuzz testing: 1) on activities, *i.e.* they continuously launch the AUT and rotate the device, 2) on resources, *i.e.* they decompile the AUT obtaining its directory structure and sources and randomly replace an image with a large one, compile the new version and run it on the emulator, and 3) on API, *i.e.* remove API calls that should apply garbage collection but that may not be correctly managing it (*e.g.*, destroy(), System.exit()).

In 2015, Deng *et al.* [Deng et al., 2015] present a mutation testing approach to test Android mobile applications, which can be used for test case design, to evaluate other Android testing techniques and to identify redundant tests and, thus, reduce them. Moreover, Deng *et al.* presented eight novel mutation operators specific for Android applications that are related to intents, event handlers, activity lifecycles and xml files (manifest, lay-

²<http://stackoverflow.com/>

out, ...). They automatically generate the mutants and, for each mutation, they generate a mutated APK. The tests are run both on the original application and on the mutated one and are, afterwards, compared.

Security

Security analyses if data is only accessible to the ones who are authorised to access it (confidentiality), if the system is capable of preventing unauthorised accesses and modifications of data (integrity), if all the events and actions are registered (non-repudiation), if an event can be traced back to the user who provoked it (accountability) and if the system can ensure the identity of a subject/resource (authenticity).

Avancini and Ceccato [Avancini & Ceccato, 2013] present an approach to test the communication between applications as a way of verifying if the AUT is susceptible to explicit intents sent by applications trying to maliciously obtain its information. Their goal is to detect intent inputs that violate the Operating System (OS) intent filter but are considered valid and whose result is accessing sensitive information. This is achieved in three phases. In the first phase, intents that would be blocked by OS intent filter are generated. In order to do this the manifest file is parsed in order to detect the intent filters permitted by the AUT; a valid implicit intent is sent in order to verify if the OS presents the AUT as a potential target of the intent; the initial intent is mutated in order to negate each of the OS' intent filters; and, finally, each of the generated intents is then sent as an implicit intent in order to ensure they violate the OS intent filter. The next phase consists in verifying if the AUT incorrectly validates the generated intents and, if so, if they access sensitive information. This consists in cloning the AUT and performing a dynamic analysis on both the original and cloned applications. The analysis is performed by instrumenting both the AUT and its clone, sending each intent to both and analysing the generated logs. Avancini and Ceccato implement this approach on a Eclipse plug-in integrated with the Android Development Tools.

Keng *et al.* [Keng et al., 2016] present MAMBA, a tool for privacy testing of Android mobile applications. Their approach starts with the decompilation of an Android APK into byte code and identify the activities that are connected to privacy sensitive API calls. The generated bytecode is then processed in order to obtain a control-flow graph of call backs using the GATOR/SOOT tool [Yang et al., 2015]. This graph is then traversed and test cases that may lead from the root activity to the previously identified ones are generated. Finally, these test cases are run on an automated front-end tester. This tester was built on top of the Automated Model-Checker (AMC) [Lee et al., 2013].

Maintainability

Maintainability analyses how modular the system is (modularity), if a component can easily be used in more than one system or to build new systems (reusability), how effective and efficient is the assessment of the impact that a modification on one component has on the other components (analysability), to which extent the system can be modified without introducing defects (modifiability) and how effective and efficiently a test criteria can be established (testability).

Gomez *et al.* [Gomez et al., 2013] identify the main problems faced by capture-replay tools as they do not correctly handle sophisticated GUI gestures, do not consider the device's sensors state and do not correctly handle the precise timing requirements of mobile applications. Therefore, they present a capture-replay tool (for Android applications), Reran, that uses the *getevent* tool provided by Android SDK to capture the low-level event stream of the device, *i.e.* it captures both the UI events injected into the application and the events provoked by the device's sensors. For the replay part of the tool they developed an agent that is able to reproduce the recorded stream of events with micro-second accuracy and to inject multiple events at the same time.

Adamsen *et al.* [Adamsen et al., 2015] present an approach to improve existing test suites by injecting them with special events that represent adverse conditions (*e.g.*, unexpected events that may arise at any time or memory shortage). The main problem they faced is the decision of when to inject the event while not disrupting the normal functioning of the AUT nor increasing the time of the test suite on a non-scalable manner. They consider three main rules to solve this: the same event should not be injected in consecutive instructions that only read data or the state of the application, events should not be injected in the middle of a sequence of events altering the state of the application, and events should not be injected after sleep instructions. As such, they inject events after instructions that trigger events (after the test has triggered the event and the event handler has completed). They implement their approach on a tool, Thor.

Yu and Takada [Yu & Takada, 2015] present an approach for generating test cases for mobile applications, considering events that are not handled by the AUT (*e.g.*, incoming phone-call, GPS turned off). Their approach is composed of five main steps: 1) a static code analyser that identifies the events to which the application reacts and in which state it needs to be in order to react to them; 2) event repository where the different events are manually classified and stored (*e.g.*, the category GPS includes the events "GPS turned on" and "GPS disconnected"); 3) event-pattern generator, which combines the events into event-patterns (an automatic formalisation of the classifications of the previous step); 4) the user is recommended some categories that classify the UI events that are not handled

by the AUT's code even though they should be. The user can then select the categories he deems important; and finally 5) test case generator, which generates test cases for each of the patterns, even if the events are not handled by the AUT.

Vieira *et al.* [Vieira et al., 2015] focus their work on applications that are required to work under certain conditions, such as one that needs to be at a certain country or that aims at panic situations in major events. Thus, they propose a Context Simulator that has two main components: a desktop one and a mobile one. The desktop component allows context modelling and the decomposition of high level context to support the simulation. This sends raw sensed data to the mobile component, which is used to test the AUT instead of the data received from the real device's sensors.

Brief Summary

When automating the testing of mobile applications it is necessary to have a model or a test oracle of the AUT. Otherwise, the testing process is prone to only finding crashes. However, the type of information the model/oracle contains differs according to the final goal: it may contain the correct behaviour of the AUT (*e.g.*, Nguyen *et al.* [Nguyen et al., 2012]), or the life cycle an application should follow (*e.g.*, Franke *et al.* [Franke et al., 2012a]) or the correct filtering of intents (*e.g.*, Avancini and Ceccato [Avancini & Ceccato, 2013]) or even information on how to classify the purpose of the application according to its description (*e.g.*, Gorla *et al.* [Gorla et al., 2014]). The output of the approaches is either a test suite (when the goal is to generate test cases) or the result of the test itself.

2.2 Reverse Engineering for Mobile Testing

The majority of reverse engineering approaches still focus on web applications, which is easily explained by taking into consideration the amount of information available on the web and the number of daily accesses to this type of applications. However, reverse engineering approaches that are applied on mobile applications are becoming more common as the users are shifting from web to mobile applications.

There are three types of reverse engineering: static, which extracts information by analysing either the source code or the byte code of the application; dynamic, which extracts information by analysing the application in runtime; or hybrid, which extracts information both statically and dynamically. When the analysis intends to analyse aspects related to the behaviour of application or how the UI responds to user action, static analysis is not the best approach due to the event-based and dynamic nature of mobile applications (more details in Section 1.1.3).

The majority of mobile reverse engineering approaches either intend to comprehend the application by obtaining models that describe it and its functionalities, *i.e.* model recovery, or to ensure its correct functioning, *i.e.* V&V, with a higher focus on the latter. In fact, even when the goal of the reverse engineering approach is model recovery, its ultimate goal is often to use these models to improve the testing process, such as Joorabchi and Mesbah [Joorabchi & Mesbah, 2012] and Yang *et al.* [Yang et al., 2013]. Joorabchi and Mesbah [Joorabchi & Mesbah, 2012] implement the iCrawler tool that dynamically exercises an iOS application and infers a model of its UI states. They state their intent of using this model to apply smoke testing to iOS applications and to generate test cases. Yang *et al.* [Yang et al., 2013] apply a hybrid approach to Android applications to obtain a model of the application. Their goal is to use this model as input to a model-based testing approach. They statically extract the set of events supported by the GUI of the AUT and then dynamically crawl the AUT to obtain the model.

The remaining of this Section describes some recent work that applies reverse engineering to mobile testing, classified according to the System and Software Quality Requirements and Evaluation ISO [ISO/IEC, 2011].

Reliability

Researchers have been using reverse engineering techniques to test the reliability of mobile applications.

In 2011, Hu and Neamtiu [Hu & Neamtiu, 2011] present their work on finding and classifying bugs on Android applications. They are able to identify and categorise three different bug categories: activity errors, *i.e.* errors in the implementation of the activity protocol; event errors, *i.e.* the action applied by the AUT did not correspond to the event fired; type errors, *i.e.* runtime type exceptions. Their approach is divided in three phases. In the first phase they automatically generate JUnit test cases from the application source code, using the Activity Testing class in JUnit. The second phase consists in identifying the events to exercise these test cases and fire them, obtaining the log files of the executions with detailed information about the application (the GUI events fired, the method calls and exceptions). In order to obtain these log files, Hu and Neamtiu use an instrumented virtual machine (VM), the Dalvik VM. Finally, in the third phase, they analyse the produced log files in order to identify and categorise the different bugs. In order to do so they identified associated patterns that indicated what the correct behaviour is to each identified class of errors (activity, event or type) enabling the execution to be flagged as correct or bug. The pattern associated to the activity error indicates the correct flow in the activity life cycle and verifies if each of the AUT's activities follows it,

the pattern associated to the event error indicates that for each event fired there should be an event handler and the pattern associated with the type error detects if a *ClassCastException* was thrown during the execution. Later, in 2013, Azim and Neamtiu [Azim & Neamtiu, 2013] presented a reverse engineering tool, A³E, which statically analyses the byte code of the application and generates execution traces following one of two algorithms: Targeted Exploration, which generates events to focus on the exploration of one activity, and Depth-First Exploration, which generates events in order to provide a thorough exploration of the AUT. Afterwards, it fires the events on the AUT running on a Dalvik VM, obtaining the log files and presenting activity and method coverage statistics. They present the A³E tool as an aid to the tester's test case generation work.

In the works by Franke *et al.* presented in Section 2.1 they focus on asserting if the life cycle of the application is correctly implemented (*e.g.*, test data persistence when an activity is destroyed and later resumed). One of the problems Franke *et al.* face is the lack of a model of the life cycle that truthfully represents the AUT as this model is often not provided and, even when the documentation includes it, Franke *et al.* have found it to be usually incorrect or incomplete. Thus, in [Franke et al., 2011] they present an approach to obtain the actual life cycle model of the AUT, which they use for their testing approach. Their approach starts by identifying the different life cycle callback-methods of the application and injecting logging information on all of them. Following, it is necessary to define test cases that trigger all the mobile platform triggers (*e.g.*, incoming call, rotation of the screen) and run them. The life cycle model is, then, inferred from the logging information obtained.

Amalfitano *et al.* start their research on mobile testing by implementing a crawling tool, A²T² [Amalfitano et al., 2011]. This tool is capable of crash testing the AUT by firing random events as well as regression testing. Moreover, it is capable of extracting the model representing the different screens and the relations between them. This tool's main limitation is the pre-definition of the technique used to derive the test cases. This limitation took Amalfitano *et al.* to work with Atif Memon in adapting Memon's GUI Ripper [Memon et al., 2003] to Android mobile applications and developing AndroidRipper [Amalfitano et al., 2012b]. AndroidRipper follows a dynamic reverse engineering approach, using Robotium [Robotium, 2016] to analyse and interact with the AUT. AndroidRipper injects events into the AUT according to the exploration algorithm the tester deems more suitable and is capable of 1) extracting the GUI Tree of the AUT (similar to the one on [Amalfitano et al., 2011]) and 2) crash testing the AUT. They use the Android Instrumentation class³ in order to identify where the failure occurred and to apply cover-

³<http://developer.android.com/reference/android/app/Instrumentation.html>

age metrics. Their approach has evolved along the years [Amalfitano et al., 2012a, 2015]. Currently, they are capable of extracting 1) an event flow graph that indicates the possible sequences of events to be fired in the AUT; 2) a state machine that represents the different states in which the AUT can be and the events that provoke a transition from one state to the other (GUI Tree); and 3) UML sequence diagrams. Moreover, they are able to 1) automatically crash test an Android mobile application; 2) generate JUnit test cases that lead to the crashes found; and 3) analyse the coverage of the testing process. AndroidRipper works in three different phases. Initially, they rip the AUT, *i.e.* they traverse the application by injecting events using Robotium, and extract a state machine representing the AUT. Secondly, they generate the different sequences of events that can be exercised in the AUT. In order to avoid an explosion of test cases they decided to have one test case for each pair of adjacent events, *i.e.* for each state they gather all the pairs $\langle \textit{previous event}, \textit{next event} \rangle$ and generate a path from the starting state to there. These test cases are on the format of JUnit test cases. Finally, these test cases can be run in order to detect crashes in the AUT. In 2014, Amalfitano *et al.* propose an approach to take advantage of the existence of patterns in order to improve the test coverage of AndroidRipper [Amalfitano et al., 2014]. However, they have yet to present the implementation and results of such approach.

In 2015, Imperato [Imparato, 2015] implement an approach combining the GUI Ripping process with Input Perturbation Testing, *i.e.* it generates inputs for the test cases generated by the GUI Ripping tool. The GUIAnalyzer, the tool in which this approach is implemented, receives the GUI Tree and the screenshots obtained by the GUI Ripping process and generates text inputs by perturbing the regular expressions associated with each input text type: removing mandatory sets, disordering the sequence of sets and inserting invalid and dangerous characters (*e.g.*, empty or long string). The result is an Extensible Markup Language (XML) file with the list of the generated inputs associated with the corresponding element.

Zhu *et al.* [Zhu et al., 2015] present an approach, Cadage, to automatically crawl the application while finding crashes and inferring a GUI model based on the application's UI states and events. Their approach is divided in four phases: inferencer, which analysis the current state of the screen, identifying the fireable events, selector, which selects the event to execute according to an algorithm developed by Zhu *et al.*, executor, which sends the selected event into the application, and modeller, which creates, and updates, the GUI model. Their approach does not require any sort of instrumentation nor access to the source code of the AUT.

This year, Moran *et al.* [Moran et al., 2016] present a crash detection tool, Crash-Scope, to report on the Android mobile applications crashes. Their approach is divided

in seven phases. First of all, they obtain the source code of the application by decompiling the APK of the AUT. Secondly, they statically analyse the obtained code in order to identify activities related to contextual features (*e.g.*, wifi). The third phase consists in crawling the application taking into consideration contextual events (*e.g.*, turn wifi on and off) on the activities identified in the previous step. During this phase CrashScope records any crash that may have occurred. Next, (in a post-exploration phase) they analyse the occurrences recorded and generate a natural language report on the traces that led to crashes. While generating the reports, additional data necessary for the reproduction of the traces is also inferred. Finally, they use the recorded information to reproduce the crash.

Ting Su [Su, 2016] propose a tool, FSMdroid, which combines reverse engineering and model-based testing in order to test Android applications. The reverse engineering process includes static analysis (using Soot [Sable, 2016]) in order to identify the possible UI events and a dynamic analysis (using A³E [Azim & Neamtiu, 2013]) to obtain a FSM of the AUT. From this model, they apply a MCMC method to mutate the model in order to guide the test generation. At the end, they execute the generated tests, finding bugs associated to mishandled exceptions and analysing the code coverage the tests provide.

Security

Along the years there have been some approaches concerning security issues on mobile applications, being the main one regarding to the permissions of the application. This is a problem both for Android and iOS applications: Android applications indicate which permissions are requested by the application but the user has to either accept them or cancel the installation and iOS applications have always full access to the device's resources (as stated in [Dar & Parvez, 2014]).

Batyuk *et al.* [Batyuk et al., 2011] focus on trying to mitigate unwanted accesses on Android applications, *i.e.* they try to identify possible security vulnerabilities, such as unwanted access of user data. In their work, Batyuk *et al.* decompile the AUT and reverse engineer its source code by applying data mining and regular expressions to the source tree obtained by the decompiling process in order to detect unwanted accesses and report them to the user. These accesses are then mitigated according to the user's security preferences by applying a patch to the decompiled binary as long as they do not impact the AUT's core functionality.

More recently, Dar and Parvez [Dar & Parvez, 2014] start by decompiling the Android application Package (APK) of the AUT and removing the unwanted permissions. This step follows the same idea as Batyuk *et al.*'s work. Next, they analyse the source code in

order to identify when accesses to the device's resource are required. Finally, they ask the user, in run time, if they want to grant it to the application whenever the sections of the source code identified in the previous phase are accessed.

Ravitch *et al.*'s approach [Ravitch et al., 2014] intends to identify undesirable communication patterns between applications. This problem arises from the range of possibilities each Android permission enables. For instance, an application with access to the contacts and which is able to send information to another application can easily send information about the contacts to a malicious application. Ravitch *et al.*'s approach is divided in two phases: 1) analyse the binary code of the application in order to retrieve a corresponding call graph from which the sensitive information access and to whom it is sent are identified and 2) construct a graph representing the communication patterns of the collection of applications with whom the AUT communicates.

Even though several approaches focus on permissions problems, there are other issues to take under consideration.

Mahmood *et al.* [Mahmood et al., 2012] improve the fuzz testing [Takanen et al., 2008] of Android applications. They are able to detect security defects related to interface exceptions (caused by direct inputs to the device), interaction exceptions (caused by communication failures with other components), permissions exceptions (caused by access violations) and resources exceptions (caused by abnormal system usage). Their approach begins by decoupling the APK of the AUT (if the source code is not available). The obtained manifest and XML layout files are used to infer the architecture model of the AUT, which represents the AUT's architecture and UI layout. The code obtained from the decoupling is then parsed. The result of the parsing is the inferring of the call graph model, which represents the possible method invocation traces, and the completion of the architecture model with the associations between the layouts and the activities. Once these models are generated, they are used to generate execution traces (following a MBT process) and the traces are executed on the AUT using Robotium [Robotium, 2016]. During the execution, the AUT is monitored according to several Android-specific monitoring facilities, which report on the different defects detected (crashes, exceptions, access violations, resource trashing). Finally, they analyse the results obtained and correlate the traces to the problems, identifying potential security vulnerabilities.

Hu *et al.* [Hu et al., 2014] propose a tool, Duet, a library integrity verification tool for Android applications, to detect security threats that arise from using third-party libraries in Android applications development. They state that using these libraries can expose the application to security threats because they can be modified to become malicious. There are even some libraries that are masqueraded in order to not look suspicious (*e.g.*, using namespaces similar to Google's to pretend to be published by Google). They divide their

approach in three phases. Initially, they obtain the original library files from the library providers, compile them into Dalvik byte code and reverse engineer this code to get the .class files. Secondly, they reverse engineer the APK of the application in order to obtain the .class files and selects the files that correspond to the libraries. In both these phases, the hash value of each file and the hash value of a merge of the files are calculated and in both the Dare reverse engineering tool [Octeau et al., 2012] is used. Finally, if the merged hash values coincide the test passes. If not, it compares the hash values for each of the files. If they all match then the test also passes.

In 2015, Soh *et al.* Soh et al. [2015] present an approach to detect Android application clones in the different Android markets. Most approaches attempting this use static reverse engineering techniques [Chen et al., 2014; Zhou et al., 2012]. However, Soh *et al.* stated that these techniques were unreliable due to most applications being obfuscated and, thus, opted to make use of dynamic techniques. Their approach is divided in four phases. Initially, they obtain the manifest file from the application's APK and identify the different activities of the app. Secondly, they run the application on an emulator exercising the different activities and extract their view hierarchy into an XML file using UIAutomator [Developers, 2015a]. The third phase aims at identifying dynamic software birthmarks. In order to do so, they parse the XML files and generate the corresponding birthmarks: a vector where each element represents the frequency count of a unique combination of the view class (*e.g.*, `FrameLayout`, `Button`), selected attribute (except package, index, bounds, text, resource-id and content description as they are easily manipulated) and value of selected attribute. The fourth and last phase consists in the clone identification, which implies analysing millions of applications. They compare the generated vectors from different applications with the original one by solving it as a near neighbour problem, with the help of the E2LSH tool [Andoni, 2005] mixed with the Hungarian algorithm [Kuhn, 1955] to solve the activity assignment problem (how to know which pairs of activities should be compared). This approach can be used to identify applications which just want a share of the success of the original application as well as to identify malicious applications that pretend to be the original application.

Maintainability

Anand *et al.* [Anand et al., 2012] present an approach to obtain the inputs necessary to efficiently dynamically analyse an Android application. They state that the effectiveness of a dynamic approach is utterly connected to the input events used when crawling the AUT. They apply concolic execution [Sen et al., 2005], also known as dynamic symbolic execution, to infer paths necessary to explore the application. Their main goal is to correctly

identify which events should be fired and what should be their input (*e.g.*, the coordinates of a tap in the screen).

Jensen *et al.* [Jensen et al., 2013] present a method that reverse engineers mobile applications in order to improve the code coverage of the automated tests. The main problem with automatic testing that Jensen *et al.* identify is their inability of reaching parts of code that are “protected” by strong constraints. Their main goal is to identify a sequence of events that are able to access that code block. Their approach requires access to the source code and to an UI model of the AUT and is divided in two separate phases. The first phase processes the application’s source code by performing concolic execution of each event handler to infer path conditions and symbolic states for the paths. In the second phase, the tester indicates which part of the code was not reached during the testing process and the tool uses the information retrieved from the first phase and a model representing the User Interface of the application to infer the sequence of events that should be executed to access the indicated part of the code.

Machiry *et al.* [Machiry et al., 2013] present a tool, Dynodroid, that is capable of exploring an Android application. Even though the objective of the tool is not to test the application but to explore it as thoroughly as possible, its output is the traces of the exploration which can be used to ease testing. They follow a (novel at the time, 2013) observe-select-execute approach in which they cyclically execute events, observe the result on the UI and select a new event to fire. The main challenges of their approach were the selection of the event and how to observe the UI changes. To solve the former they present three different selection strategies: 1) selecting an event that has not been often selected (frequency); 2) selecting an event randomly (UniformRandom); and 3) selecting an event based on the context of the application and on the results of its prior executions (BiasedRandom). To solve the latter they use an instrumented Software Development Kit (SDK) in order to identify the callbacks (when a listener is called) and the ViewServer service to obtain the hierarchy and layout of the UI.

Choi *et al.* [Choi et al., 2013] propose an automatic mobile testing algorithm, SwiftHand, that generates sequences of test inputs by combining active learning with testing and focuses on interface testing. One of Choi *et al.*’s main concerns is to reduce to a minimum the amount of application restarts, which is usually a problem with active standard learning algorithms. Thus, SwiftHand tries to continuously explore the application following a path that does not require restarting the application. Along the exploration, SwiftHand iteratively builds the model of the AUT, an extended deterministic labelled transition system. Each event selection depends on the model inferred by SwiftHand so far, trying to quickly maximise branch coverage.

Fratantonio *et al.* [Fratantonio et al., 2015] present another tool, CLAPP, that detects

loops in Android applications which are “one of the most useful and essential constructs when writing programs” but are “one of the most challenging to handle [when applying program analysis]”. Moreover, in the context of Android applications “loops within the main application are opaque to the framework” and require “performance-intensive, fine-grained dynamic monitoring systems”. The CLAPP tool, on the other hand, statically analyses the byte code of the application by applying techniques such as loop identification, backward data-flow analysis on use-def chains, selective abstract interpretation and code reachability analysis. Fratantonio *et al.* are able to extract information on how the loops are controlled and on their body and behaviour. Their goal is for CLAPP to be used to ease the task of identifying (in)correctly implemented loops.

Brief Summay

Mobile testing approaches mostly focus on functional sustainability, reliability, security and maintainability. The main limitation of the approaches using reverse engineering techniques is the lack of prior knowledge about the AUT. This restricts the aspects of the AUT that can be tested and is the reason why no approach focuses functional sustainability testing. Thus, most approaches focus on detecting crashes or unwanted accesses. In order to provide a more thorough analysis, instrumentation is generally used either on the AUT or on the emulator (*e.g.*, the Dalvik VM). Instrumenting the AUT means the application needs to be pre-processed before the testing process and instrumenting the system means that a device/normal emulator can not be used. In both cases, the runtime execution of the application is affected.

2.3 Pattern-Based Testing

One of the most well-known UI Patterns is the login/password. In this case the problem is the necessity of identifying who is trying to access some parts of the application or certain information and the solution is requesting the unique pair login/password to the user before enabling them to move further. Mobile applications are no exception to the presence of patterns as proven by studies such as Erik Nilsson’s [Nilsson, 2009] and Sahami Shirazi’s [Sahami Shirazi et al., 2013]. Back in 2009, Erik Nilsson [Nilsson, 2009] identified some recurring problems when developing an Android application and categorised the UI design patterns that could help solve them. For instance, he presents the “Change the screen orientation” design pattern in order to solve the need to “horizontal scrolling” problem and the “let the keyboard cover part of the UI” and “Only use the part of the screen that will not be covered by the keyboard” design patterns to solve the “Handling

crowded dialogs when software keyboard is shown and hidden” problem. More recently, in 2013, Sahami Shirazi *et al.* [Sahami Shirazi et al., 2013] studied the layout of Android applications trying, among other goals, to verify if those layouts presented any patterns. They identified two types of patterns: 1) a layout container that contains other layout elements as well as widgets, such as a Scroll View containing a Linear Layout; 2) layout elements that only contain widget elements, such as a Linear Layout containing two Text Views or a Linear Layout containing an Image View and Text View. This study was conducted taking into consideration a static analysis of the layout and its elements. Moreover, Theresa Neil [Neil, 2014] studied and compiled different UI Patterns that can be found in mobile applications.

Considering the presence of patterns in mobile applications, several approaches try to take advantage of them in order to ease mobile testing. The main difference between the different approaches is the type of patterns considered.

Batyuk *et al.* [Batyuk et al., 2011] define a pattern as a set of malicious accesses, *i.e.* they detect unwanted accesses and privacy leaks in the applications. They apply pattern mining to the source code of the applications taking into consideration the user’s security preferences.

Hu and Neamtiu [Hu & Neamtiu, 2011], on the other hand, associate the notion of pattern with possible failure classes, *i.e.* typical places where failures can occur. They identify crashes related to the mishandling of the application’s life cycle (activity errors), to the mishandling of events like the lack of an event handler (event errors) or to exceptions in type casting (type errors).

More recently, Amalfitano *et al.* [Amalfitano et al., 2014] present three types of patterns: GUI patterns, event patterns and model patterns. GUI patterns relate to the presence of GUI widgets (*e.g.* Edit Texts or Buttons) or of complex interaction structures (*e.g.*, sorting list or sharing options). Event patterns represent a sequence of contextual events that exercise the application (*e.g.*, loss and successive recovery of GPS signal while walking or Network instability). Model patterns are submodels that can be detected in the GUI model that contain mistakes or false assumptions made by their crawler (AndroidRipper presented in Section 2.2).

Costa *et al.* [Costa et al., 2014] focus on defining UI Test Patterns for UI Patterns, *i.e.* they define test strategies (UI Test Patterns) to test recurring behaviour that can be found in different applications (UI Patterns), such as Login and Master Detail.

Shahriar *et al.* [Shahriar et al., 2014] identify memory leak patterns, *i.e.* they trace the memory leak problems to their cause. Some examples are memory leak through event listeners and memory leak through animation activity. These patterns are used to generate test cases that are capable of identifying memory leaks in the application.

Yu and Takada [Yu & Takada, 2015] group events into categories, which are their patterns. For instance, all the events related to an incoming call (*e.g.*, incoming call received, incoming call answered) are grouped on the same pattern and the events related to GPS connectivity (*e.g.*, GPS turned on, lost GPS signal lost) are group in another pattern.

Holl and Elberzhager [Holl & Elberzhager, 2014] present a mobile specific failure pattern classification. The failures can be of the type Behaviour, *i.e.* failures describing that stimuli to the AUT do not result in the specified output (*e.g.*, data incorrectly received/sent while communicating with the back-end), Design, *i.e.* failures related to incorrect colour, font, dimension, shape or position of GUI elements (*e.g.*, an element that is specified in the model but is not present in the application), and Content, *i.e.* a content is defined but is not present (*e.g.*, reading *Adio* instead of *Audio*). Each of these patterns can be sub-classified: Behaviour can be of the type Transition, Transaction, Dynamic Content or Event, a Design failure can relate to interaction element or style, and Content failures can be classified as static picture, static animation or static text. Holl and Vieira's failures classification consisted of applying Mauser *et al.*'s classification [Mauser *et al.*, 2013] to mobile applications (Holl was also part of this work).

Most of the patterns concern failures on the UI of the AUT and, thus, there are some similarities between them. Amalfitano *et al.*'s GUI and event patterns [Amalfitano *et al.*, 2014] can be compared to the UI Patterns presented by Costa *et al.* [Costa *et al.*, 2014] as they represent behaviour commonly observed on mobile applications and Costa *et al.*'s UI Test Patterns can be compared to Holl and Elberzger's patterns [Holl & Elberzhager, 2014] as they illustrate how a failure on the AUT can be detected. Even though Hu and Neamtiu [Hu & Neamtiu, 2011] focus on detecting failures, they are more focused on the faults that originated them. Yu and Takada [Yu & Takada, 2015], on the other hand, just classify the type of events that can be fired.

The only approaches that do not relate to patterns within the UI are the ones of Batyuk *et al.* [Batyuk *et al.*, 2011] and Shahriar *et al.* [Shahriar *et al.*, 2014] as they shift their focus away from the UI and into malicious intents and memory leaks, respectively.

2.4 Android Mobile Test Automation Frameworks

In order to develop a mobile test automation tool, it is necessary to analyse the existing mobile test frameworks. Considering the implementation (see Chapter 4) will focus on Android application, there are non-official and official frameworks to consider.

Non official frameworks

Even though Google has provided a testing framework after the launch of Android in 2008, that framework was very basic and did not fulfil the community's testing necessities. Thus, the community has developed some testing frameworks on its own. The most popular one is Robotium.

Robotium [Robotium, 2016] is an Android test automation framework that was introduced in January 2010. It offers a simple testing API that works on any device regardless of the Android version it runs and, thus, it rapidly spread within the community.

Robotium offers several advantages when automating tests:

- it is capable of testing both native and hybrid applications;
- it does not require access to the source code of the AUT;
- it can be used for any version of Android;
- it can access and modify the status of the device's sensors and services.

However, it also presents some disadvantages:

- the name of the AUT's main activity must be hard coded into the test code;
- it can not read the contents of the screen;
- it can only access and modify the status of the services for which the AUT asked permission.

In 2012, Calabash [Xamarin, 2016], another test automation framework for both Android and iOS, was released. The main advantages it presents are:

- its syntax is based on easy to understand natural language;
- it can access and modify the status of different sensors and services;
- it reports on the performance (*e.g.*, CPU, memory) of the device.

Its main disadvantages are:

- it requires access to the source code of the AUT;
- it is not capable of reading the contents of the screen.

Appium [Appium, 2016] is a widely used open-source testing framework that first appeared on April 2013 [SourceLabs, 2013]. Its main advantages are:

- it supports both Android and iOS applications by being built on top of Google's UIAutomator and Apple's UiAutomation, respectively (for Android versions prior to 4.2 it uses the Selenium Web Driver);
- it supports native, hybrid and mobile web applications;
- it supports different languages and frameworks;
- it does not require access to the source code of the AUT;
- even though the AUT's package name and main activity must be known, they can be provided as input, *i.e.* they do not have to be hard coded into the test code;
- it can access and modify the status of different services and sensors.

On the other hand, it also presents some disadvantages:

- it is incapable of returning the screen contents;
- elements' localisation still needs improving, *e.g.*, it is not possible to locate an element based on its *id*;
- it is still not stable, occasionally producing some unexpected errors.

Finally, Selendroid [Foundation, 2016] was also released on 2013 as Selenium [OpenQA, 2016] for Android, *i.e.* it is a test automation framework for Android applications. Its main advantages are:

- it supports native and hybrid applications;
- it does not require access to the source code of the AUT;
- the localisation of elements on the screen is simpler than with Appium;
- it presents an inspector tool to read and analyse the contents of the screen.

Nevertheless, it also presents some disadvantages:

- the inspector tool does not offer an API and, thus, it can not be used programmatically, *i.e.* it is incapable of reading the contents on the screen;
- it can not access nor modify the status of the device's sensors and services.

Official frameworks

Even though there has always been an official testing API, it was not until recently that Google started to actually invest time and effort in this: in November 2012, Google released the first version of UIAutomator that provided cross-application testing [Developers, 2012]; in December 2014 Google released Espresso 2.0 that was a fully working testing framework focused on a single application [Developers, 2014]; and later on March 2015, a new version of UIAutomator was released [Developers, 2015b].

One can think of Espresso as the long awaited official version of Robotium:

- it is capable of identifying the elements on the screen in order to interact with them but it is unable of obtaining a list of the screen's content;
- It is instrumentation-based and thus needs to know beforehand which application will be tested;
- it can only read and modify the status of the services identified on the AUT's manifest.

Even though it was only introduced in Android 4.2, it can be used on devices running Android 2.2 or higher. One of the main advantages of Espresso is that it provides automatic synchronisation of test actions with the UI of the AUT, *i.e.* it is not necessary to wait some time within interactions to ensure the UI has already responded because Espresso is able to detect when the AUT's main thread is idle (unlike Robotium).

When it was introduced, UIAutomator [Developers, 2015a] was a limited version of Espresso that enabled cross-application testing on devices running Android 4.1 or higher, *i.e.* it was not associated with the AUT and, thus, could interact with other applications other than the AUT. However, it was not capable of reading and modifying the status of services of the device, unless if the tester would program it to simulate how a user would do it (*e.g.*, open Settings application, go to Wifi menu, turn Wifi on). This implied having to leave the application in order to turn a service on and off and it had to take into consideration that different devices could have the Settings application organised in different ways (even within the same Android's version manufacturers usually build their own version of the system on top of the original Android) [Developers, 2015a].

On March 2015, Google launched a new version of UIAutomator, UIAutomator 2.0 along with Android 5.1. The main difference of this version compared to the previous one is that it is based on instrumentation while keeping its main property of being cross-platform. As such, from this point on, UIAutomator was able to access the UiAutomation API⁴ that is based on the Accessibility Service and, thus, it is able to read the

⁴<https://developer.android.com/reference/android/app/UiAutomation.html>

content of the screen (its elements and their properties). Moreover, bringing together the instrumentation-based API with the cross-application property means it is capable of reading and modifying the status of every service of the device, not only the ones used by the application, as well as getting information on the the device's sensors. The main disadvantage of this framework is that it only works on devices running Android 4.3 or higher.

The main disadvantage of both Espresso and UIAutomator is that they only work on native applications.

Comparison of Frameworks

Apart from UIAutomator all frameworks have some similarities:

- they are instrumentation-based;
- they are associated with a single application;
- the device's services they can access depend on the AUT's permissions;
- they are not capable of reading the elements of the screen nor providing information about them.

They also present some distinguishing characteristics:

- Unlike Robotium and Appium, Espresso is not able to interact with non-native applications;
- Neither Robotium nor Appium are capable of detecting when the AUT responded and thus need to add some waiting time after each interaction, while Espresso is able to detect when the AUT has responded;
- only Robotium works for all Android versions;
- Espresso is an official framework and, thus, it is always in accordance to the latest Android revisions.

The two main problems of these frameworks are 1) it is necessary to know beforehand which AUT is going to be tested; and 2) it is not possible to analyse the screen in order to automatically decide which event to fire.

2.5 Main Conclusions

When defining test strategies it is important to define what should be tested and how, *i.e.* define what is the correct behaviour of the application and what is its actual behaviour. The majority of the testing approaches that focus on the UI response of the application, either demand prior knowledge of the alleged behaviour of the application or are only capable of detecting crashes.

Reverse engineering can be extremely useful to solve the problem of verifying what is the actual behaviour of the application under test. However, most approaches require access to the source code, even if just to instrument it to obtain extra information.

The main advantage of patterns is that they relate a problem to a working solution. Therefore, it should be possible to define the correct behaviour for a certain situation. In fact, there are already some works that try to associate patterns with known failures.

The approach presented in this document intends to reverse engineer an application in order to know what its actual behaviour is, while at the same time analysing if that behaviour is consistent with patterns defined from design and test guidelines. Thus, this work does not involve dealing with source code in any of its steps, being one of the few that present a completely dynamic instrumentation free approach, and presents a catalogue of patterns that work as a test oracle defining what the correct behaviour of the application should be. Moreover, the vast majority of mobile GUI testing works uses GUI testing to test the AUT but does not focus specifically on testing the AUT's GUI unlike our approach, which focuses on testing the GUI of the application.

Chapter 3

Pattern Based Testing

Chapter 1 defined the research hypothesis as “Mobile applications have generic recurring behaviour, independent of their specific domain, that may be tested automatically by combining reverse engineering with testing within an iterative process”.

This Chapter proposes an approach to verify this hypothesis and it is structured as follows. Section 3.1 presents some definitions important for the correct comprehension of the approach. Section 3.2 presents a general overview of the approach described in the following sections. Section 3.3 presents the current state of the catalogue of patterns. Sections 3.4, 3.5 and 3.6 present the three main aspects of the approach. Section 3.7 describes the different artefacts produced by the approach. Section 3.8 draws some conclusions.

3.1 Definitions

This Section defines some concepts that are useful for understanding the approach. Moreover, some important distinctions between similar concepts are also presented. The terminology here presented is valid throughout the whole document.

Screen

A screen is an internal representation of the contents of the device’s display at a certain moment in time. It is defined as a hierarchical tree, *i.e.* a directed graph in which any two vertices are connected by exactly one path. Each vertex has exactly one parent (except for the root of the tree, which has none) and may have zero or more children. Each vertex of the tree represents an element of the device’s display structure. These elements may be visible to the end user (*e.g.*, a button) or just used to define the layout of the visible elements, *i.e.* to group elements and define their positioning on the screen. Henceforth, the term screen is used as a simplification of the *device’s display*.

Screen versus Activity

According to Android's documentation¹, an activity is “an application component that provides a screen with which users can interact in order to do something”. However, one activity may consist of several screens as the content being displayed may change even if the activity remains the same. For instance, two different screens are necessary to display two different newspaper articles even though they are both handled by the same activity, the one whose purpose is to display an article.

Element versus Widget

As previously mentioned, the elements represented by the vertices of the tree representing the screen may be visible to the user or just used to define the screen's layout. Moreover, it may or may not be possible to interact with the elements (click, edit, ...). A widget is:

- a) a visible element such as a button, a check box or a text;
- b) a structural element with which it is possible to interact. For instance, in Figure 3.1 the clickable item is not the one representing the text but its parent (*TwoLineListItem* in Figure 3.1b).



Figure 3.1: List corresponding layout in the TomDroid application

Event

An event in a mobile application can be of two types:

- an UI event, *i.e.* an event fired by the user when interacting with the application, such as tapping a widget (also known as clicking on a widget);

¹<https://developer.android.com/reference/android/app/Activity.html>

- a system event, *i.e.* an event fired externally to the application, such as the detection of a new available network, an incoming call or message and the modification of the orientation of the phone (from portrait to landscape, for instance).

Different widgets may enable different UI events: whereas an edit text box may be edited, a button may only be clicked. Firing UI events enables the user to navigate throughout the application. Nevertheless, the application should also be able to correctly handle the system events.

Execution trace

An execution trace is defined as:

$$ExecutionTrace : (S_1, a_1, S_2, a_2, \dots, S_n), n > 1 \quad (3.1)$$

, *i.e.* an alternate sequence of states (S) and actions (a). An execution trace represents a possible exploration of the application. In this context the state contains the information of the device and the information of all the elements visible on the screen at a certain moment. The action may be an event or an observation of the state of the screen.

Pattern

As defined in Chapter 1, a pattern is “an idea that has been useful in one practical context and will probably be useful in others” [Fowler, 1997]. More simply, it can be defined as a recurring solution to a recurring problem.

In the context of this work, a pattern corresponds to recurring UI layout or recurring behaviour that can be found in mobile applications. More formally, it is a restriction on part of the execution trace with $n \geq 1$, in which n is the number of states on that sub-trace. This sub-trace can be in the format of $S, (a, S)^*$, *i.e.* it has at least one state and if an action occurs a new state has to follow. It is structured as a tuple $\langle Goal, P, V, A, C \rangle$, in which (considering S_i the initial state and S_j the final state):

Goal is the ID of the pattern;

P is a restriction on S_i , *i.e.* the precondition (boolean expression) defining the conditions in which the pattern should be applied

V is a set of pairs [variable, value] relating input data with the variables involved. These must be instantiated by the user at the start;

A is the sequence of actions to perform, which may be empty ($j = i$, *i.e.* there is no action and, thus, exactly one state), simple ($j = i + 1$, *i.e.* exactly one action occurs and,

thus, there are exactly two states) or complex ($j > i + 1$, *i.e.* two or more actions occur resulting in three or more states);

C is a restriction on S_j and eventually on S_i and the intermediate observations, *i.e.* it is the set of checks to perform (also known as the post-condition).

Applying the GOF form described in Section 1.1.4, the goal (G) represents the *Name*, the pre-condition (P) represents the *Problem*, the actions (A) and variables (V) represent the *Solution* and the checks (C) represent the *Consequences*.

In other words a pattern can be formally defined as

$$Goal[configuration] : P \rightarrow A[V] \rightarrow C \quad (3.2)$$

, *i.e.* for each configuration of a goal $Goal[configuration]$, if the pre-condition (P) is verified, a sequence of actions (A) is executed with the corresponding input values (V). In the end, a set of checks (C) is performed.

In this work, two types of patterns are considered: UI Patterns and Test Patterns. The purpose of an UI Pattern is to define when and how to detect the presence of the pattern. Regardless of its name, a UI Pattern may be associated either to the GUI of the application, such as the presence of the Action Bar, Side Drawer or Login/Password, or to system events, such as rotating the screen, receiving an incoming call or losing wireless connectivity. A Test Patterns is a generic test strategy that verifies if the corresponding UI Pattern is correctly implemented.

Taking the Login/Password Pattern as an example, the UI Pattern should detect when a Login/Password form is present on the screen (*e.g.*, there must be a text box for the user name, a text box that expects a password for the password and an *ok* button) and there would be two corresponding Test Patterns which would verify if when a correctly user name/password pair is provided the login is successful and if it is unsuccessful otherwise (the correct pair would have to be provided by the user).

Both UI and Test Patterns share the same definition (the tuple $\langle Goal, P, V, A, C \rangle$). However, the meanings of each of the tuple's elements have slightly different interpretations depending on the type of pattern. In an UI Pattern, P defines when to verify if the pattern exists, A defines the actions to execute in order to verify if the pattern is present and C validates the presence of the UI Pattern. On the other hand, in a Test Pattern, P defines when the test should be executed, A defines the sequence of actions to execute in order to test if the corresponding UI Pattern is correctly implemented and C works as the test oracle and indicates if the test passes or fails, *i.e.* whether or not the UI Pattern is correctly implemented.

3.2 General Overview

In order to validate the research hypothesis, it is necessary to define an approach that is capable of 1) exploring a mobile application extracting information on its behaviour and 2) testing recurring behaviour, *i.e.* patterns, on the fly. This Section provides a general overview on this approach, whose main purpose is to improve mobile testing automation.

In order to define the approach a catalogue of patterns must be compiled. This catalogue contains the UI patterns to be identified and the test strategies associated to each of them, *i.e.* the Test Patterns, and can be used to test any mobile application. The catalogue defined so far can be found in Section 3.3 and is useful as a proof of concept but it should never be considered final. In fact it should be continuously improved and extended (to include new interaction trends, for instance).

Figure 3.2 depicts the components diagram of the approach.

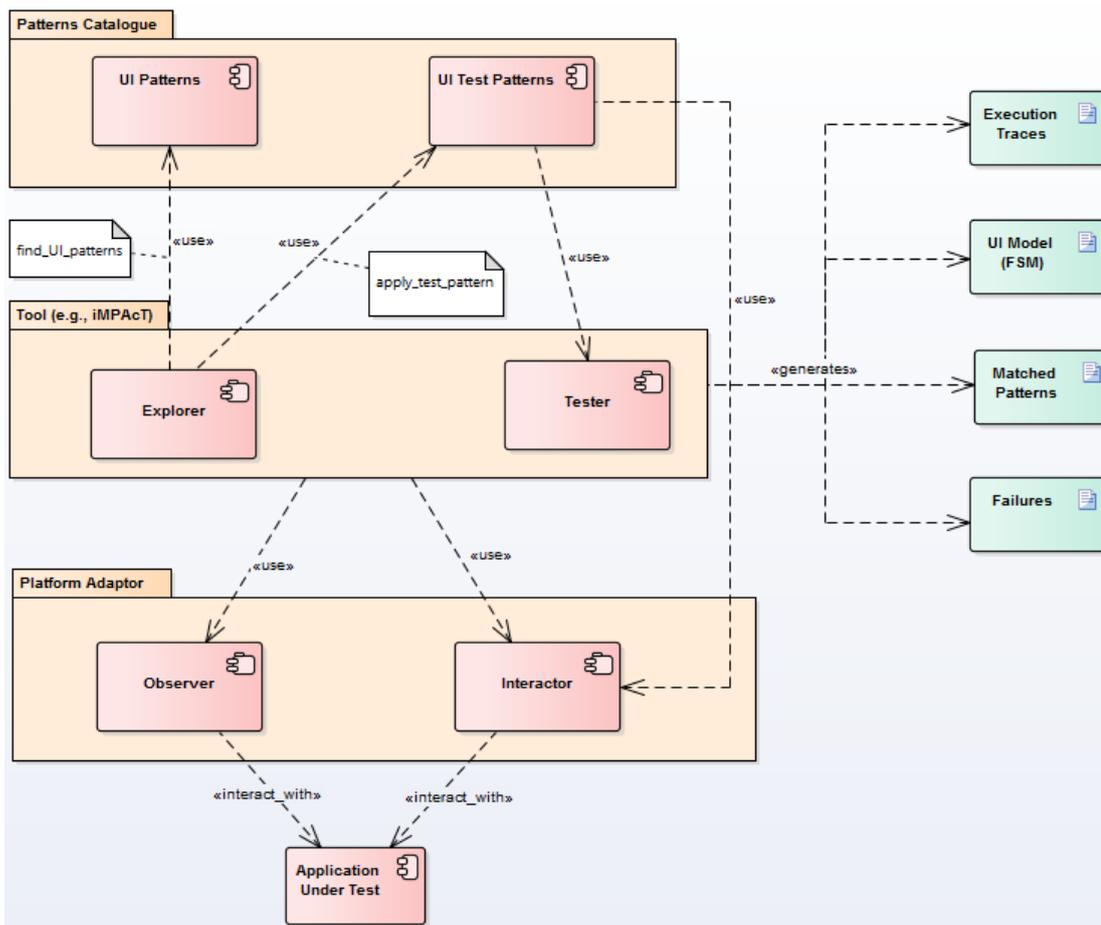


Figure 3.2: Components Diagram of the Approach

The approach is an iterative process that is divided in three main components: *Patterns Catalogue*, *Tool* and *Platform Adaptor*.

The *Patterns Catalogue* contains the set of UI Patterns to be identified in the application and the corresponding Test Patterns. This catalogue can be used to test any application. The user only needs to indicate which patterns should be identified and tested during a specific run.

The tool consists of two main components: the *Explorer* and the *Tester*.

The *Explorer* applies a dynamic reverse engineering process to automatically explore the AUT, *i.e.* it works like a crawler injecting events onto the device and receiving a response from it that is analysed.

The *Tester* consists of methods used to test the UI Patterns found on the application, *i.e.* methods used to apply test strategies that verify if the UI Patterns found during the exploration are correctly implemented.

Whenever the *Explorer* sends an event to the application, it verifies if there currently are any UI Patterns present in the application in a pattern matching process (*find_UI_patterns*). If one is detected, the *Explorer* accesses the corresponding Test Patterns (read from the catalogue) and applies them to the application under test (*apply_test_pattern*). Applying the checks of a Test Pattern uses methods defined in the *Tester* and a report is produced.

Finally, the *Platform Adaptor* is responsible for bridging the *Tool* with the AUT, *i.e.* it depends on the operating system (OS) on which the AUT is being run. It consists of two main components: the *Observer* and the *Interactor*. The *Observer* obtains information on the state of the application and the *Interactor* injects events onto the device/application. Both *Observer* and *Interactor* are used by the two components of the *Tool* and the *Interactor* is also used when applying the actions of the Test Patterns.

At the end of the process, four main artefacts are produced:

- Execution log: the log of the exploration, *i.e.* the sequence of events fired into the application;
- UI Model: a finite state machine representing the application (its different activities and how to navigate between them);
- Matched Patterns: which of the UI Patterns were identified in the application;
- Failures: which patterns were not correctly implemented *i.e.* which Test Patterns failed.

The user only needs to indicate which application they want to test and which UI Patterns should be identified and tested.

3.3 Catalogue of Patterns

One of the most crucial aspects of this approach is the catalogue of patterns as it defines what will be tested and how. This catalogue consists of a set of UI Patterns (UIP) and the corresponding Test Patterns (TP), *i.e.* the test strategies.

This Section describes the pairs UI Pattern - Test Patterns currently defined in the catalogue. Even though most patterns are present in all OS, the type of solution proposed can vary as different OS follow different design guidelines. For example, in Android tabs should be drawn on the upper part of the screen whilst in iOS they should be drawn on the bottom.

The patterns presented in this Section are mainly based on the guidelines provided by Android on how to design applications [Android, 2015c] and on how to test them [Android, 2015a]. However, the testing and design guidelines from iOS can also be used to define other patterns. The solution for each of the implementation challenges defined for each of the patterns is described in Chapter 4.

Two aspects common to all Test Patterns is that 1) one of their preconditions must be the presence of the UI Pattern they are testing and 2) one of their pre-conditions must be that they have not yet been applied to the activity being analysed (except for the second TP of the Orientation Pattern as explained in Section 3.3.2).

3.3.1 Side Drawer Pattern

The navigation among the different screens and hierarchy of a mobile application can be done in several forms. One of these is the *Side Drawer* (or Navigation Drawer) UI Pattern [Android, 2015b; Neil, 2014], *i.e.* a transient menu that opens when the user swipes the screen from the left edge to the centre or clicks on the app button (button on the left of the application's *Action Bar* as depicted in Figure 3.3).

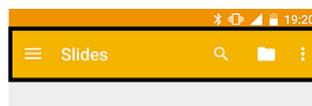


Figure 3.3: Action Bar of the Google Slides application

Figure 3.4 depicts an example of this UI Pattern.

According to Android's guidelines, a correctly implemented Side Drawer should occupy the full height of the screen.

The UI Pattern verifies if there is a hidden side drawer and is defined as:

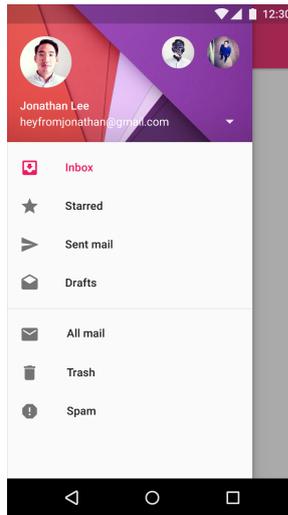


Figure 3.4: Example of the side drawer UI Pattern [Android, 2015b]

Goal: "Side Drawer exists"

P: {"true"}

V: {}

A: [observation]

C: {"side drawer exists and is hidden"}

The corresponding TP opens the side drawer and verifies if it covers the full height of the screen. Hence, it is defined as:

Goal: "Side Drawer occupies full height"

P: {"UIP present *and* side drawer available *and* TP not applied to current activity"}

V: {}

A: [open side drawer, observation]

C: {"covers the screen in full height"}

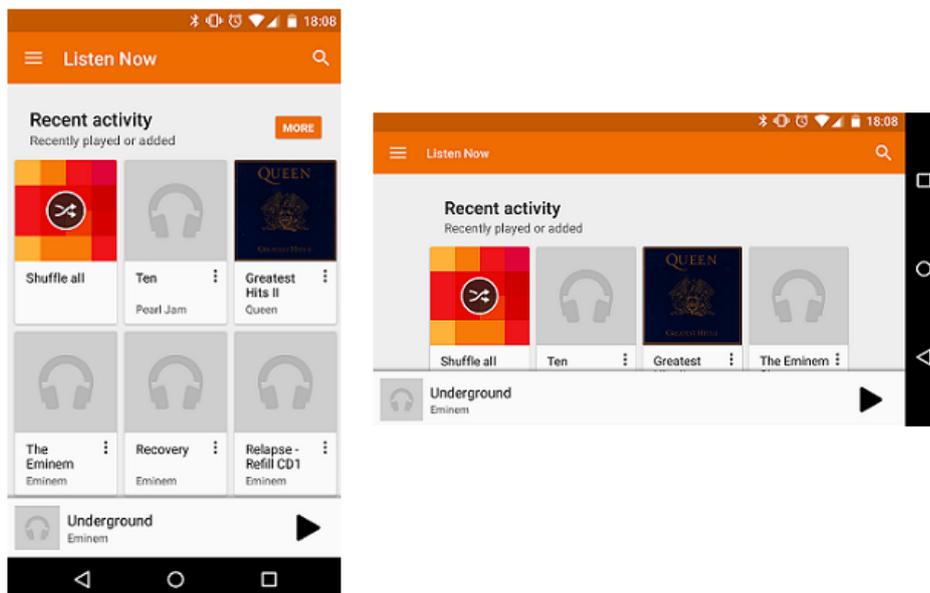
The subtrace of the execution trace considered in this pattern is (*State before opening the Side Drawer, Event that opens the Side Drawer, State after opening the Side Drawer*). The checks are a restriction only on the last state.

The main challenge of implementing this pattern lays on how to identify the side drawer element.

3.3.2 Orientation Pattern

Mobile devices have two possible orientations: portrait and landscape, as depicted in *a* and *b* of Figure 3.5, respectively.

When rotating the device, the screen of the application also rotates and its layout is updated. However, according to Android's Guidelines for testing [Android, 2015a] there are two main aspects the developers should test: 1) no user input data should be lost, *i.e.* all the content that the user has entered, such as text in *text edit*, or checking a *checkbox*, is still present after the rotation, and 2) widgets should not disappear when rotating the screen. Figure 3.5 depicts a rotation of the screen in which some widgets present in the first screen (Figure 3.5a) seem to disappear from the the second one (Figure 3.5b). However, this is not a failure, the items were simply dislocated due to lack of space. In order to avoid incorrectly characterising this situation as a failure, after rotating the device, the screen is scrolled and the new elements are added to the screen.



(a) Portrait orientation

(b) Landscape orientation

Figure 3.5: Example of rotation in the Google Play Music application

The UI Pattern verifies if it is possible to rotate the screen and is defined as:

Goal: "Rotation is possible"

P: {"true"}

V: {}

A: [observation]

C: {"it is possible to rotate the screen"}

There are two corresponding Test Patterns. The first one is applied when the UI Pattern is present and this TP has not yet been applied to the current activity and it rotates the screen and verifies if the widgets that were present in the screen before rotation are still present. If the element is not found after the rotation, the screen is scrolled to obtain the missing elements. This TP is defined as:

Goal: "UI main components are still present"

P: {"UIP is present *and* TP not applied to current activity" }

V: {}

A: [observation, rotate screen, observation, scroll screen, observation]

C: {"widgets still present" }

The second Test Pattern is applied when the UI Pattern is present, user inserted data on an element and this TP has not yet been applied after inserting data in that element. It rotates the screen and verifies if the data the user inserted is still present. Alike the previous TP, if a widget is not found, the screen is scrolled to obtain the missing elements. This Test Pattern is defined as:

Goal: "Data unchanged when screen rotates"

P: {"UIP is present *and* user data was entered *and* TP not applied to this element" }

V: {}

A: [observation, rotate screen, observation, scroll screen, observation]

C: {"user entered data was not lost" }

The comparison of the before and after rotation screens reports an error when one of the following situations occurs:

1. one of the screens is a pop-up and the other is not;
2. one of the screens has a side drawer and the other does not;
3. a widget is present on the first screen but not on the second one;
4. user inserted data is present on the first screen but not on the second one.

If one of the first two aspects is detected there is no need for comparing the elements. If a side drawer is detected in both screens, only the elements contained within it are compared.

The sub-trace of the execution trace considered in this pattern is (*State before rotating the screen, Event that rotates the screen, State after rotation, Event that scrolls the screen, State after rotation and scroll*). The checks are a restriction on the first and last states.

The main challenge of implementing this pattern is the match between the initial screen widgets (before rotation) with the second screen widgets (after the rotation). This is difficult because elements do not have a unique ID.

3.3.3 Resource Dependency Pattern

Several mobile applications use external resources, such as GPS or Wifi. Moreover, several of these are dependent on the availability of those resources. As such, it is important to verify if the application does not crash when the resource is suddenly made unavailable as stated by [Android, 2015a].

The corresponding UI Pattern verifies if the resource in question is being used by the AUT and is defined as:

Goal: "Resource in use"

P: {"true"}

V: {"resource", resource_name}

A: [observation]

C: {"resource is being used by the AUT"}

The corresponding Test Pattern verifies if the application crashes when the resource in question is made unavailable. It is defined as:

Goal: "Application does not crash when resource is made unavailable"

P: {"UIP && TP not applied to current activity"}

V: {"resource", resource_name}

A: [observation, turn resource off, observation]

C: {"application did not crash"}

The resource_name may be wifi, 3G signal, GPS or Bluetooth.

The subtrace of the execution trace considered in this pattern is (*State before turning resource off, Event that turns resource off, State after turning resource off*). The checks are a restriction only on the last state.

The main challenge of this pattern is to know when a resource is being used and how to identify if the application crashed.

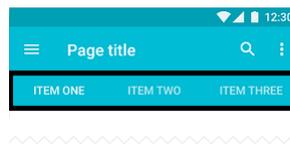


Figure 3.6: Example of a set of tabs

3.3.4 Tab Pattern

In some applications, such as Facebook or Twitter, it is possible to find tabs. A tab (Figure 3.6) “makes it easy to explore and switch between different views or functional aspects of an application or to browse categorised data sets” [Android, 2016d].

There are some design guidelines regarding tabs [Butcher & Developers, 2016; Butcher & Nurik, 2016] that should be followed to improve user experience: 1) there should only be one set of tabs so that it is obvious which content is being displayed; 2) the position of the tabs on the screen should follow the respective OS guidelines: in Android applications tabs should be localised on the upper part of the screen (“Android’s tabs for view control are shown in action bars at the top of the screen” [Android, 2016b]) and in iOS applications they should be on the lower part of the screen (“A tab bar always appears at the bottom edge of the screen” [Developer, 2016]); and 3) swiping the screen horizontally should only change the selected tab, *i.e.* there should be no element enabling horizontally scroll apart from the one handling the tabs.

The corresponding UI Pattern verifies if the screen displays any tabs and is defined as:

Goal: “Presence of Tabs”

P: {“true”}

V: {}

A: [observation]

C: {“There are tabs present”}

There are three corresponding Test Patterns. The first TP verifies if there is only one set of tabs present on the screen and is defined as:

Goal: “Only one set of patterns”

P: {“UIP && TP not applied to current activity”}

V: {}

A: [observation]

C: {“there is only one set of tabs at the same time”}

The second TP verifies if the tabs are correctly positioned on the upper part of the screen and is defined as:

Goal: "Tabs position"

P: {"UIP && TP not applied to current activity"}

V: {}

A: [observation]

C: {"Tabs are on the upper part of the screen"}

Finally, the third TP swipes the screen horizontally in order to verify if the selected tab changes and is defined as:

Goal: "Horizontally scrolling the screen should change the selected tab"

P: {"UIP && TP not applied to current activity"}

V: {}

A: [observation, swipe screen horizontally, observation]

C: {"the selected tab changed"}

Two different subtraces of the execution trace are considered. For the first two test patterns the subtrace is (*State in which tabs are present*) and the checks are a restriction on this state. On the other hand, for the last test pattern, the subtrace of the execution trace considered is (*State before swiping the screen, Event that swipes the screen, State after swiping the screen*). The checks are a restriction on the first and last states.

The main challenge of implementing this pattern is to identify if the selected tab actually changed as sometimes developers do not correctly update the *selected* property.

3.4 Exploration

The first of the three phases of the approach is the Exploration, whose responsibility belongs to the *Explorer* of Figure 3.2. The exploration consists of four aspects:

1. analyse the current state of the application;
2. identify the events that can be fired;
3. decide which event to fire;
4. fire the selected event.

The remaining of this Section explains each of these steps in detail.

Analyse the current state of the application

The first step of the exploration is to analyse the current state of the application. This consists in a recursive reading of the different screen elements (starting in the root and stopping in the leaves).

Each element of a screen has different properties, which must also be identified in this phase. The main properties are:

- *position (or bounds)*: indicates the position of the element within the screen as a rectangle. For instance, *Rect(0,0,10.0,1776)* with the bounds of a full screen element on a screen whose width was 1080 and whose height was 1776. This property is mandatory;
- *class*: identifies the type of the element (*e.g.*, button, text, check box, spinner). This property is mandatory;
- *resource_id*: identifies the element. Depending on the OS this property may or may not be unique. For instance, in Android it is similar to the *class* in CSS files². Thus, regardless of its name, this property is not considered to be neither unique nor mandatory;
- *text*: text that is printed on the screen within that element. This property is not mandatory and is usually associated with buttons, text and edit text elements. Its value may be an empty string;
- *content description*: text that is associated to the element but is not visible on the screen and that usually describes the result of clicking on that element (*e.g.*, a button with the symbol of a magnifying glass has no *text* associated but its content description is usually a variation of “Search”). This property is not mandatory and is usually associated with buttons that present an image instead of textual information;
- *package name*: every application has a unique name (the name of the package) and this property indicates the application to which the element belongs (the package name of the application);
- *enabled*: indicates whether or not it is possible to interact with this element. A disabled (not enabled) element is usually depicted in light grey tones. This property is mandatory.

²https://developer.mozilla.org/en-US/docs/Web/CSS/Class_selectors

Moreover, each element has also some mandatory boolean properties indicating if they enable the different events to be fired on them (*clickable, checkable, editable, scrollable, long-clickable, focusable*) as well as some mandatory boolean properties that indicate its state (*checked, focused, selected*). Finally, the *password* property indicates whether or not the element represents a password.

Identify the events that can be fired

According to their properties, it is possible to map the elements with executable events. For instance, it is possible to fire the *click* event on a clickable element or to *edit* an editable element. The same element can even have more than one event associated, *e.g.*, an edit text can usually be clicked, edited or long-clicked, *i.e.* pressed for a period of time instead of immediately lifting the finger.

Currently, the following events are considered: click, check, long click, edit and scroll. Moreover, in some Android applications prior to Android 5.0, the menu is only accessible by firing the *press device's menu button* (depicted in Figure 3.7). Therefore, when this button is present this event is also added.



Figure 3.7: Example of a menu button (on the right)

Decide which event to fire

The decision of which event to fire can follow different algorithms, such as depth-first or breath-first. In this work, three different algorithms are considered:

1. *execute once*;
2. *priority to not executed*;
3. *priority to not executed and list items*

Execute only once

In this algorithm each event is only fired once, *i.e.* if an event has already been fired then it is no longer considered.

Priority to not executed

In this algorithm, the not yet executed events are the first to be considered (alike the *execute only once* algorithm). If at a certain moment all available events have already been executed, the algorithm considers the events that lead to a different screen which still has not yet executed events. This bit of information is collected when the event was previously executed and is updated whenever the event is executed. To avoid possible cycles, each event can only be executed a set amount of times.

Priority to not executed and list items

This algorithm is similar to *Priority to not executed* because it first tries to execute all the not yet fired events and, afterwards, considers the events whose execution leads to a screen that still has not executed events. However, in both phases of this algorithm, the events that are executed on items that belong to a list have priority towards the remaining events. The main advantage of this strategy is the attempt of further exploring the content of the screen before changing screens. The “click on the App button” event may have one of two results: opening the side drawer (see example on Figure 3.4) or going back to a previous screen. As both actions result in a change of screen (once the side drawer is opened it will most likely navigate to a different part of the application), this event is only considered after no more events are available.

No Restrictions

There is yet a fourth exploration algorithm which has no restrictions and is, thus, not relevant.

Final Decision

All these algorithms define the level of priority an event has over the other. When two events have the same priority, the decision of which of those events to fire is random.

When there are no available events, the *back* button of the device is pressed in an attempt to return to the previous screen.

Fire the selected event

After the screen is completely analysed and the event is decided, the final step of the exploration phase consists in firing the event. More information on this can be found in Chapter 4.

3.5 Pattern Matching

After an event is fired, it is necessary to verify if any of the UI Patterns defined in the catalogue is present. This implies analysing once again the current state of the screen and consists in verifying if its *pre-condition* holds, executing the *actions* necessary and verifying if all *checks* are met. If so, the UI Pattern is considered found.

3.6 Testing

Whenever an UI Pattern is identified, the corresponding Test Patterns are retrieved from the patterns catalogue. This phase consists in applying each of these patterns on the current state of application, *i.e.* if the *pre-condition*, which must include the presence of the UI Pattern, holds, then the *actions* are executed and the *checks* are verified. If a Test Pattern fails, it means that the corresponding UI Pattern is not correctly implemented and a report is produced. If no UI Pattern is found in the Pattern Matching phase then this phase is skipped.

Once all the Test Patterns are applied, the exploration phase is resumed.

3.7 Artefacts

At the end of the exploration two main artefacts are produced: the report of the exploration and a model of the behaviour observed during the exploration.

3.7.1 Report

The report produced is divided in two parts: 1) Log of the exploration and 2) result of the tests.

3.7.1.1 Log of the Exploration

The first part of the report (log of the exploration) is also divided in two parts: a) identification of the AUT and of the parameters of the exploration and b) sequence of events fired during the exploration.

Identification of the AUT and of the Parameters of the Exploration

The AUT is identified by its package name and the parameters of the exploration are:

- the version of the OS running on the device/emulator on which the AUT is being run;
- the screen's dimensions, *i.e.* height and width;
- the y coordinate at which the application starts: unless an application is full screen it will not start being drawn from the top of the screen.

Sequence of Events Fired During the Exploration

Each of the events fired on the application is logged as a tuple $\langle Event, Element, Executions \rangle$, where *Event* identifies the event being fired, *Element* identifies the element on which the event is fired and *Executions* indicates the number of times the event was fired on the element:

Event is a tuple with the type of the event it represents (*e.g.*, click, edit) and the input value used on the event. The input value is optional depending on the type of event. For instance, a click does not require any input value but an edit requires the value to be introduced; and

Element is optional depending on whether the event was fired on an element (*e.g.*, click on a button) or not (*e.g.*, rotate screen). *Element* is a tuple $\langle class, resource_id, text, content_description, bounds \rangle$ as described in Section 3.4.

Before logging an event, the screen on which it will be fired is identified with an *id*.

Whenever a Test Pattern fails, the corresponding UI Pattern is identified and reported as incorrectly implemented and the reason reported (*e.g.*, “rotation is not correctly handled: some components disappear when rotating”) would be the text when the first Test Pattern of the Orientation UI Pattern (see Section 3.3.2) fails.

The current state of the catalogue (presented in Section 3.3) enables the detection of seven different types of GUI Failures (the *.top* and *.bottom* refer to the y coordinates of the corresponding element in the screen, being $top < bottom$):

- side drawer not correctly drawn on the screen: being e_{sd} the element representing the side drawer and h the height of the screen, this failure indicates that $e_{sd}.bottom < h \vee e_{sd}.top > 0$;
- an item of the screen disappeared, *i.e.*, an item like a dialogue or a menu option disappears when rotating the screen: being E_p the set of elements of S_p , the screen on the portrait position, and E_l , the set of elements of S_l , the screen on the landscape position, this failure indicates that $\exists e \in E_p : e \notin E_l$;

- user input data disappeared when rotating the screen: being P_p and P_l the set of pairs $(property, value)$ for the input properties (text, content description, selected, checked) of E_p and of E_l , respectively, this failure shows that $\exists p_1 \in P_p, p_2 \in P_l : p_1.property = p_2.property \wedge p_1.value \neq p_2.value$;
- more than one set of tabs available on the screen: being T the set of sets of tabs in the screen, this failures indicates that $T.size > 1$;
- tabs are incorrectly positioned on the screen: being t the only set of tabs present on the screen, this failure indicates that $t.bottom \geq h/3$, being h the height of the screen (the first third of the screen is considered to be its upper part as explained in Section 4.8.4);
- swiping the screen when on the presence of tabs does not change the selected tab. There are two ways of defining this failure:
 - a) the selected tab is the same before and after swiping the screen, *i.e.* being t_1 and t_2 the set of tabs before and after swiping the screen, respectively, this failure indicates that $\exists e_1 \in t_1, e_2 \in t_2 : e_1.equals(e_2) \wedge e_1.selected = true \wedge e_2.selected = true$;
 - b) the content under the set of tabs does not change after swiping: being E_1 and E_2 the sets of elements under the set of tabs before and after swiping the screen, respectively and t_1 and t_2 the set of tabs in the screens before and after swiping, respectively, this failure indicates that $\exists e_1 \in E_1 : e_1.top \geq t_1.bottom \wedge e_2 \in E_2 : e_2.top \geq t_2.bottom \wedge e_1.equals(e_2)$;
- application crashes (either related to the testing or to the exploration of the AUT): being $S.E$ the set of elements of screen S this failure indicates that $\exists e \in S.E : e.package = "android" \wedge e.isPopUp()$ (as defined in Section 4.7).

Results

The report also contains the following results:

- the number of events executed and the number of events identified (and the corresponding percentage);
- if each of the UI Patterns was detected and, if so, whether or not it was correctly implemented. If a failure is detected the report indicates how many times it happened;

- the duration of the exploration.

This report can be use to reproduce the exploration while also reproducing the failures. For instance, after correcting a failure, the tester may reproduce the logged trace to verify if the failure still occurs.

3.7.2 Model of the Observed Behaviour

The reverse engineering process obtains, at the end of the exploration, a model of the application, a GUI model, which is the form of a FSM and represents the behaviour of the application. This model is a directed graph that represents distinct states of the application and the transition between these states: $G = \langle Screens, Events \rangle$, where *Screens* is the set of unique screens (activities) and *Events* is the set of events that have been executed during the exploration. Each screen S is defined as $S = \langle R, E \rangle$, where R represents the root element of the screen and E the set of elements present on the screen. An element e is defined as $\{(property, value)\}$, *i.e.* an element is a set of pairs of properties and the corresponding values. A simple example is depicted in Figure 3.8. The arrow identified as $e1$ connecting state $S0$ with state $S1$ represents the event that being fired in the activity represented by state $S0$ takes the exploration to the activity represented by state $S1$ of the application.

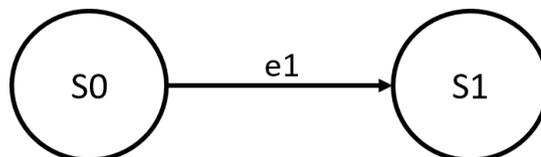


Figure 3.8: Example of a finite state machine

There can be three types of states being represented in the model:

- an activity of the AUT. The state representing this activity is identified as *Screen* $\langle i \rangle$, in which i is the id of the activity (corresponds to the order in which the activities were reached during the exploration). Screen 0 is, thus, the initial state of the FSM;
- a crash. When the application crashes such is recorded and a state representing it is added to the FSM. This state is identified as *crash Screen*. Only one state of this type may be represented in the model. If more that one event leads to a crash, they will all connect to this state in the FSM;

- the exploration left the application. Whenever an event makes the exploration leave the application, a state is added to depict this situation. This state is identified as *out of AUT Screen*. Only one screen of this type can be represented in the model. If more than one event makes the exploration leave the AUT, they will all connect to this state in the FSM.

The model generated is currently useful for aiding the user to comprehend the application and, together with the report, it may be used to understand where the failures occur. Moreover, in the future, it may be used to prove properties about the application using model checking like in [Coimbra Morgado et al., 2012]. As a way of improving the comprehension of the FSM, the tester may choose to collect screen shots of the different screens obtained during the exploration (option selected in the window of Figure 4.2) and, thus, replacing the state's representation for the corresponding image on the FSM.

As stated in 3.1, two or more screens may represent the same activity. Therefore, even though the term *Screen* is used on the model to simplify, each state of the FSM corresponds to an activity of the AUT. If the tester chooses to use screenshots of the AUT to represent the states, a screenshot of the first screen identified for an activity is used.

3.8 Conclusions

This Chapter proposed an approach to verify the research hypothesis defined in Chapter 1. The approach follows an iterative process that interacts with a mobile application, looks for UI Patterns and tests the ones found *i.e.* applies the corresponding Test Patterns. These patterns (both UI and Test) are defined in a catalogue which is reusable for any application and that can always be improved by adding new patterns.

At the moment the catalogue contains four patterns: side drawer, orientation, resource dependency and tab.

This approach brings together the advantages of crawlers and model based testing approaches as it presents a dynamic exploration while the testing process is based on patterns defined on the catalogue. The combination of these techniques makes it possible to test an application in a fully automatic process that does not require any prior knowledge of the AUT (neither an UI model nor a test oracle are needed) and is still not limited to finding crashes as it also detects other types of failures.

Chapter 4

Implementation

This Chapter describes the implementation of the approach defined in Chapter 3. It is divided in nine sections. Section 4.1 presents the selected operating system in which the AUTs are run. Section 4.2 present the technological requirements to implement the approach and selects the best framework. Section 4.3 describes the overall organisation of mobile applications of the selected OS. Section 4.4 describes the process of defining the heuristics presented in the remaining Chapter. Section 4.5 presents the iMPAcT tool in which the approach was implemented. Section 4.6 presents the implementation of each of the phases of the approach. Section 4.7 presents some details of iMPAcT's implementation. Section 4.8 presents the implementation of some of the patterns present in the catalogue. Section 4.9 draws some conclusions and presents the limitations of the tool.

4.1 Operating System

Even though the patterns presented in Section 3.3 follow the Android design and testing guidelines, the approach can be implemented in any mobile operating system (Android, iOS, Windows). However, in order to validate the research hypothesis it is not necessary to implement the approach for all of them. The Android OS was selected for two main reasons: 1) it is currently the most widely used operating system in mobile devices [Gartner, 2016b] and 2) the vast majority of research focuses on Android applications.

4.2 Technological Requirements

As stated in Section 2.4, there are several frameworks that can be used to automate testing in Android applications. However, each of them has different specifications. In order to

consciously choose a framework it is necessary to identify the main technological requirements for implementing the approach. This Section presents this study.

First of all, the approach lays on the automatic crawling of the AUT. Hence, it is necessary to ensure the technology used is capable of accessing the contents of the screen as well as interacting with the elements within it. However, this is not sufficient as several applications require special permissions in order to fully function, such as internet or GPS access. If these accesses are not ensured it may not be possible to thoroughly explore the AUT or even run it. Consequently, it is also important that the technology is able to identify the permissions requested by the AUT as well as to read and modify the status of the corresponding services.

Thirdly, it is also necessary to consider that implementing the patterns may entail different accesses and actions making it important to choose a technology that provides a wide range of interactions with both the AUT and the device and its services and sensors.

Finally, a selling point of the approach is that it is not thought for a specific application ensuring the tool does not have to be modified or adapted to the AUT. Hence, the technology can not require the identification of the application to be hard coded.

In summary, the technology to be used for the implementation must gather the following characteristics:

1. being able to read the screen, *i.e.* identify the elements that are on the screen and their properties (class, clickable, position, ...)
2. being able to identify the hierarchy of the elements;
3. being able to identify the element to interact with, *i.e.* select the element using the elements' properties extracted when reading the screen;
4. being able to interact with the selected element;
5. being able to read and modify the status of the device's services and sensors;
6. not requiring the AUT to be identified in the code of the tool;
7. not requiring access to the AUT's source code.

In Section 2.4, several frameworks currently available in the market were described. Tables 4.1 and 4.2 present how each of those frameworks fulfil each of the requirements.

Even though neither Robotium nor Appium are capable of accessing the contents of the screen, there are some workarounds in order to adapt them to get this access.

Neither Selendroid, Calabash, UIAutomator 1.0 or Espresso are capable of accessing the screen contents and, thus, are deemed unsuitable for the implementation. Moreover,

Table 4.1: Matching of the non-official frameworks with the approach requirements

Requirements	Robotium	Appium	Selendroid	Calabash
1				
2				
3	X	X	X	X
4	X	X	X	X
5	X	X		X
6		X	X	
7		X	X	

Table 4.2: Matching of the official frameworks with the approach requirements

Requirements	UIAutomator 1.0	Espresso	UIAutomator 2.0
1			X
2			X
3			X
4	X	X	X
5			X
6			X
7	X	X	X

Espresso's purpose is to test a single application hence forcing it to be identified. Plus, it does not provide access to the status of the device's services or sensors.

Unlike all the other frameworks, Calabash even requires access to the source code of the application in order to localise the screen elements.

On the other hand, UIAutomator 2.0 is the more complete framework considering the listed requirements. Nevertheless, using this framework entails that only native applications can be targeted. There is also some information UIAutomator 2.0 is not able to extract. For instance, toast messages are not detected and there is no property providing the colour of the elements. To get a fuller understanding on the type of information accessible by UIAutomator 2.0, the reader can access the *uiautomatorviewer* tool provided by Android SDK. An example of applying this tool on the Google Calendar application is depicted in Figure 4.1. On the left is the screen exactly as it appears on the device and the right part is divided in two: on the top is the tree of the application and on the bottom are some of the properties of the element selected (the App Button, in this case).

UIAutomator 2.0 uses UiAutomation, which relies on the accessibility APIs, to obtain the contents of the screen along with their properties. This work uses four main classes of UIAutomator:

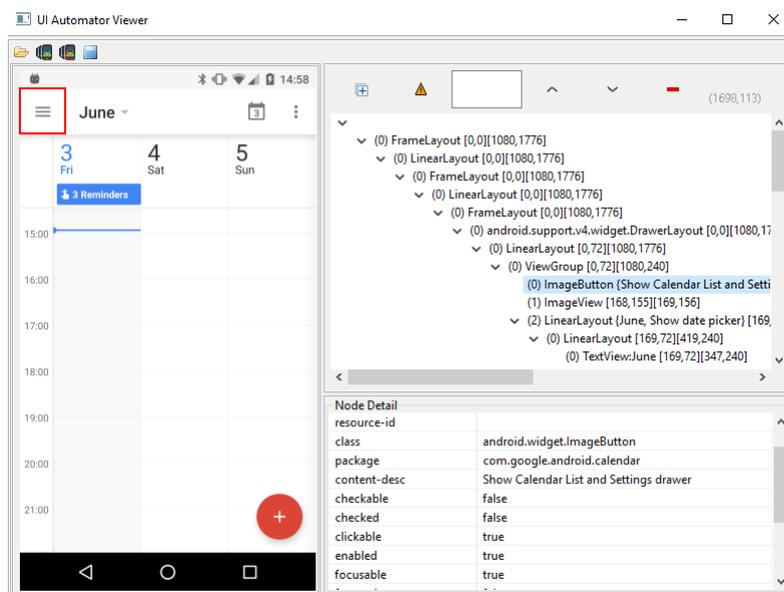


Figure 4.1: Example of the *uiautomatorviewer* tool on selecting the App Button of the Google Calendar application

- *UiObject*¹, which represents an UI element;
- *UiScrollable*², which represents an UI scrollable element;
- *UiSelector*³, which defines a filter of the object to be found in the current state of the application. It is used with *UiScrollable*;
- *BySelector*⁴, which is similar to *UiSelector* but specific for *UiObject*.

In conclusion, UIAutomator 2.0, henceforth UIAutomator, is the framework selected to implement the approach.

4.3 Android Mobile Applications

Since the release of Android 4.2 every Android application has two components: the code for the application itself (the app component) and a test project (the test component), *i.e.* a developer should implement both the code of the application and the code to test that specific application.

The app component contains:

¹<https://developer.android.com/reference/android/support/test/uiautomator/UiObject2.html>

²<https://developer.android.com/reference/android/support/test/uiautomator/UiScrollable.html>

³<https://developer.android.com/reference/android/support/test/uiautomator/UiSelector.html>

⁴<https://developer.android.com/reference/android/support/test/uiautomator/BySelector.html>

- the graddle file, which indicates the Android version of the application and its id;
- the manifest file, which indicates the package name of the application and the permissions necessary for the correct functioning of the application and it declares the different activities of the application, indicating the main one;
- the code of the application.

The test component is a JUnit 4.0 project that defines unit tests for the application defined in the app component.

4.4 Process for Defining the Heuristics

In this work a set of heuristics were implemented (comparisons of screens and elements and identification of sets of elements, such as the presence of a side drawer and tabs).

The definition of the heuristics that identify a certain set of elements followed this process:

- select twenty applications that contain the set to be identified;
- use *uiautomatorviewer* to obtain the layout of the elements and their properties;
- identify common aspects between the elements in the different applications.

The definition of the heuristics to compare elements followed this process:

- analyse the properties of each of the elements on a screen;
- compare the properties of each of the elements;
- identify the properties that could distinguish an element from the others.

This was applied for all the elements of the screens of twenty applications.

The definition of the heuristics to compare screens followed a similar process. However, the analysis focused on the differences between the screens instead of just the differences between the elements.

Furthermore, these heuristics were refined throughout the implementation.

4.5 The iMPAcT Tool

The implementation of the approach resulted in a tool named iMPAcT (Mobile PAttern Testing).

This tool is divided in two parts: the configurator, that handles the configuration of the process, and the tester, that implements the approach itself. Both were implemented in Java. However, the first part consists in a third-party application whereas the second one is an Android test project.

4.5.1 iMPAcT's Configurator

Before exploring and testing the application, the user must provide some input information to the iMPAcT tool. For this purpose, a GUI was defined. Figure 4.2 depicts the main window of the application responsible for this configuration process.

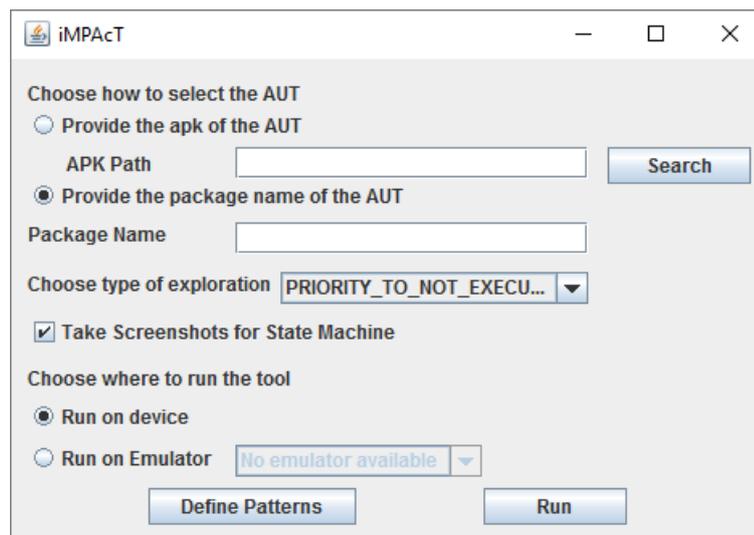


Figure 4.2: Main window of the iMPAcT tool's GUI

In this window, the user must:

- identify the application to be tested. This can be done either by providing the APK file (*Provide the apk of the AUT*) of the application (using the *Search* button) or the package name of the application (*Provide the package name of the AUT*). This second option implies that the application is already installed. On the other hand, if the APK is provided the AUT is installed regardless of whether or not it was already installed on the device/emulator;

- choose the exploration algorithm (*Choose type of exploration*): `execute_once`, `priority_to_not_executed` or `priority_to_not_executed_and_list_items`;
- choose whether iMPAcT should be run on a device that is connected to the computer (*Run on device*) or on an emulator (*Run on emulator*). In order to choose this option the user must have previously started an emulator. If more than one emulator has been started, the user can choose which one to use. In Figure 4.2, no emulators are available.
- choose to take (or not) screenshots of the AUT while it is being explored (*Take Screenshots for State Machine*). If the user chooses to take screenshots the FSM that is generated at the end of the execution will display the screenshots as a representation of the different states.

Figure 4.2 presents the default choices.

Moreover, the *Define Patterns* button gives access to a new window, depicted in Figure 4.3, in which the user can define which UI Patterns are going to be identified and tested and in which order. In this example, the side drawer and orientation patterns were chosen.

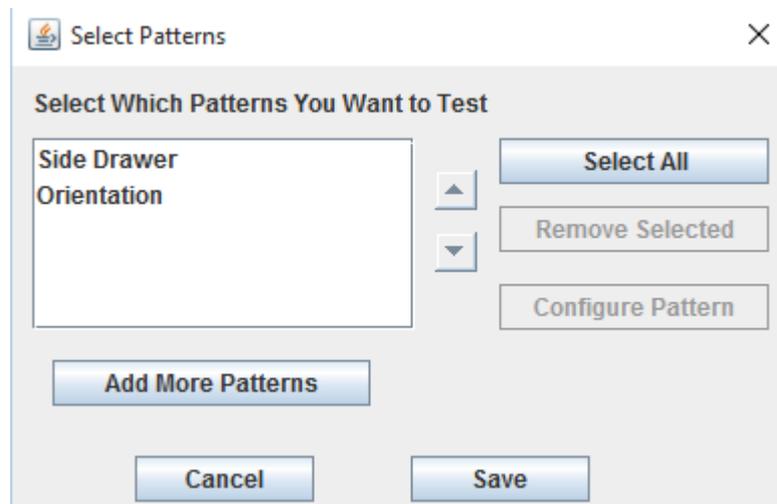


Figure 4.3: Window to define the order in which the patterns will be tested

The order in which the patterns will be tested is the order in which they appear in this window (in this case the Side Drawer pattern will be applied first and the Orientation Pattern second). This order can be altered using the *up* and *down* arrows. Selected patterns can be removed (*Remove Selected*) and more patterns can be added (*Add More Patterns*).

The *Add More Patterns* button opens yet another window, depicted in Figure 4.4, which presents all the not yet chosen patterns. The user can select one or more patterns and then click on the *Add Selected* button.

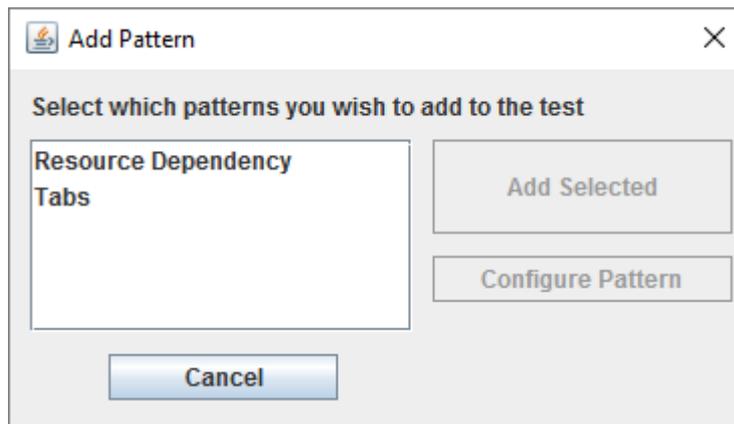


Figure 4.4: Window to select the patterns

Both Figures 4.3 and 4.4 display a *Configure Pattern* button. This button is only enabled when the selected pattern requires extra configuration, *i.e.* when it is necessary to provide values for the variables (V in the pattern definition). An example of such is depicted in Figure 4.5 where the resource dependency pattern is being configured (for *Wifi* in this case).

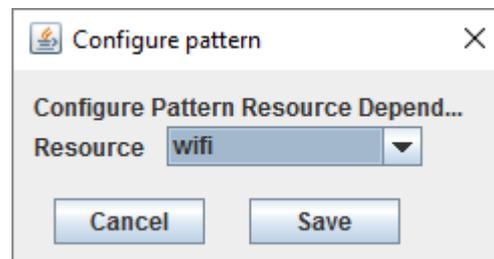


Figure 4.5: Window to configure the Resource Dependency Pattern

If no patterns are selected, iMPAcT will function just like a crawler: it will explore the application extracting its FSM and detecting if any crash occurred.

Finally, clicking on the *Run* button on the main window (Figure 4.2) installs the AUT (if necessary) and the iMPAcT tester on the chosen device/emulator (henceforth, device) and runs the iMPAcT's tester on the AUT.

4.5.2 iMPAcT's Tester

The iMPAcT tool's tester is implemented as an Android test project and it is the component that actually implements the approach. Henceforth, it will be mentioned just as iMPAcT.

In Section 4.3, an Android project was defined as having two components: the app and the test. In the case of iMPAcT the source code for the application does not exist: the app component only contains the Android Manifest and Graddle files. The list of permissions requested by iMPAcT must be defined in the manifest file and should not be confused with the ones requested by the AUT. The current permissions requested by iMPAcT are: *INTERNET*, *ACCESS_NETWORK_STATE*, *ACCESS_Wifi_STATE*, *CHANGE_Wifi_STATE* and *WRITE_EXTERNAL_STORAGE*. The source code of the implementation of the approach should be on the test component.

iMPAcT's main class is implemented as a JUnit 4.0 test project, which must contain at least three methods: *setUp()*, which is executed before the tests, *tearDown()*, which is executed after the tests and at least one test method. Although there usually are several test methods (one for each unit test), iMPAcT is defined as:

- *setUp()*: reads the configuration file produced by the configurator and sets the package name, the exploration algorithm and whether screenshots should be taken. Furthermore, it turns on the services requested by the AUT;
- *test()*: sets the catalogue as defined in the configuration file, starts the AUT and calls the method that handles the exploration/testing cycle;
- *tearDown()*: creates the final artefacts.

4.6 iMPAcT's implementation

This Section presents the main algorithms used throughout the implementation of the approach described in Chapter 3.

As previously explained, it consists in an iterative process of three phases: exploration, in which the screen is analysed and an event is selected and fired, pattern matching, in which the tool verifies if any of the UI Patterns defined in the catalogue of patterns is currently present on the screen, and testing, which takes place if a UI Pattern is detected and in which the corresponding Test Patterns are identified and applied in order to test if the UI Pattern is correctly implemented. Afterwards, the exploration is resumed.

Code 4.1 shows the general implementation of this approach.

Code 4.1: iMPAcT tool execution algorithm

```

set exploring to true
while exploring=true
  call explore
  call find_UI_patterns
  for each UI pattern found
    call apply_test_pattern
  endfor
  read exploring
endwhile

```

Each cycle of the algorithm consists in calling *explore*, which handles the exploration, followed by *find_UI_patterns*, which handles the pattern matching, and finally *apply_test_pattern*, which handles the testing and is called whenever a UI Pattern is found.

The following subsections present the algorithms implemented for each of these phases.

4.6.1 Exploration

The first of the three phases is the Exploration, which is of the responsibility of the *Explorer* of Figure 3.2. This phase consists of four steps: analysis of the current state of the application, identification of the events, decision of which event to fire and finally firing the selected event.

Analyse the current state of the application

As explained in Section 3.4, the state of the AUT is defined by the set of elements present on the screen and their properties at a certain moment. The internal representation of this screen is a tree equivalent to the one observed with the *uiautomatorviewer* tool (see Figure 4.1).

As UIAutomator makes use of UiAutomation, it is possible to obtain a tree of *AccessibilityNodeInfos*⁵ as the screen being currently displayed. This class represents “a node of the window content as well as actions that can be requested from its source”.

The following methods are available and used:

- *UiAutomation.getRootInActiveWindow()* to obtain the *AccessibilityNodeInfo* representing the root of the screen being currently displayed;

⁵<http://developer.android.com/reference/android/view/accessibility/AccessibilityNodeInfo.html>

- *AccessibilityNodeInfo.getChild(i)* to obtain an *AccessibilityNodeInfo* that represents the i^{th} child of a given node.

The last two methods are used to iteratively access all the nodes of the tree in a depth-first algorithm.

Code 4.2 presents the implementation of the screen's analysis, in which each of the elements (child) is a node in the tree.

Code 4.2: Obtain the current screen of the AUT

```
function get_current_screen
  read root
  init child to the first child of the root
  while child exists
    add child to the parent
    set child to the child_of_child
  endwhile
```

Even though the *AccessibilityNodeInfo* can be used to read the contents of the screen, it can not be used to internally store this information as it always points to the current state of the node it represents, *i.e.* the properties are updated on the fly (if a variable is instantiated with an *AccessibilityNodeInfo*, when a property of the node it represents changes, the variable is also updated) and if an element becomes inaccessible the corresponding node also becomes inaccessible.

Therefore, iMPAcT uses three internally defined classes to represent the information it stores (whose class diagram is depicted in Figure 4.6):

- *Element*, which represents each of the elements on the screen and has a corresponding *AccessibilityNodeInfo* and a set of properties. An *Element* can either be a widget or a structural element. Structural Elements have children, which are *Elements*. All *Elements* have a parent except for the root element of the screen, which is necessarily a structural element;
- *Event*, which represents an event that can be fired. It has a name, which indicates the nature of the event (click, edit, ...) and an optional input parameter (*e.g.*, the input value of an *edit* event). Additionally, it points to the screen that is obtained after the event is fired (this property can only be instantiated after the event has been fired). An *Event* may be applied on different *Elements* (*e.g.*, a click event may be applied to multiple *Elements*) and it may not be associated with any *Element* (system event);

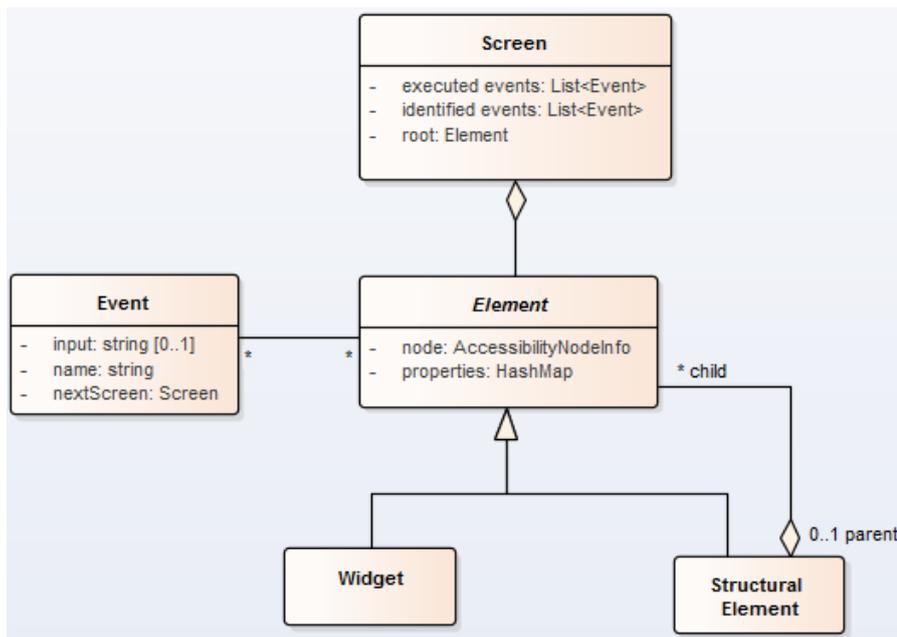


Figure 4.6: Class Diagram of the Data Structure

- *Screen*, which represents each of the screens of the application and contains the set of *Elements* that are part of the corresponding screen. It also includes a list of the *Events* that have already been identified and the ones that have already been executed.

Whenever a screen is read during the exploration, a *Screen* instance is created and its *Elements* are instantiated.

Identify the events that can be fired

Events can be fired on UI elements or on the screen (system events). This means that apart from the UI events, it would be possible to define a click event for each coordinate of the screen. As such, when identifying the events that can be fired, the only system event that is considered is *click the device's menu button* (see Figure 3.7) as in some applications prior to Android 5.0, the menu is only accessible by pressing the *menu button*. If an application can not handle this event, as it happens with the majority of recent applications, then firing this event has no consequences in the application's behaviour.

The UI events that can be fired are directly related to its element properties: if an element is clickable then the click event can be fired on it, if an element is editable then the edit event can be fired on it, if an element is scrollable then the scroll event can be fired on him, and so on.

Currently, the following UI events are implemented: *click*, *check*, *long click*, *edit* and *scroll*.

In the case of the *edit* event, the input property has to be provided. At the moment the element is analysed in order to verify if a number or a text string should be inserted. If a number is requested, the value of the input is “6”. Otherwise it is “example”. Even though at the moment these values are hard-coded it could be possible for the user to indicate other ones in the configuration step.

Decide which event to fire

The decision of which event to fire depends on the algorithm chosen in the configuration process depicted in Figure 4.2. All these algorithms, described in Section 3.4, have one aspect in common: it is necessary to match a currently identified event with the event that was previously identified in order to know if it has already been executed. This is achieved by verifying if the screen has already been visited and, if so, matching the elements of the previously visited screen with the ones that are currently being displayed. These comparisons are described in detail in Section 4.7.

Even though the *scroll* event is considered, it is only fired when it is the only possible event, *i.e.* iMPAcT fires the events on the current screen and when no more events are possible it scrolls down the screen to access new elements.

When no more events are available, an extra system event is selected, *click the device’s back button*, in order to return to a previous state of the exploration or eventually leave the AUT.

Fire the selected event

After selecting the event, it needs to be fired. If it is a system event, the methods of UIAutomator’s class *UiDevice* can be used: *UiDevice.pressMenu()* and *UiDevice.pressBack()* for pressing the device’s menu and back buttons, respectively.

However, if it is an UI event, it is necessary to transform the stored *Element* into an object that can be read by UIAutomator: *UiScrollable* to handle the scroll event and an *UiObject2* to handle the remaining events.

This is achieved by defining a selector (*UiSelector* or *BySelector* according to the type of element), which gathers all the properties (class, text, *etc.*) of the element to be identified to locate the corresponding executable object (*UiScrollable* or *UiObject2*, respectively).

Once the corresponding object is retrieved, iMPAcT fires the event using the corresponding method: *scroll()*, *click()*, *check()*, *setText()*, *etc.*

After firing an event, it is necessary to wait for the application to respond. iMPAcT uses the UIAutomator's method `UiDevice.waitForWindowUpdate()`, which indicates when a screen stops being updated.

Summary

The exploration phase can, thus, be summarised in Code 4.3, which implements the *explore* method of Code 4.1.

Code 4.3: Exploration

```
function explore
  call get_current_screen
  for each node in screen
    call node.get_possible_events
  call choose_event
  call fire_event
```

When the event being fired is *press device's back button*, the application can return to a previous state of the exploration or it can leave the application reaching the home screen. At this point, iMPAcT produces the final artefacts and stops the execution.

4.6.2 Pattern Matching

After an event is fired, pattern matching takes place, *i.e.* the tool tries to identify which UI Patterns are present on the application under test at that moment. This corresponds to the `find_UI_patterns()` method of Code 4.1. In order to reduce the exploration time, this phase is skipped if the screen after the event is exactly the same as the one before the event.

This method is similar to the `apply_test_pattern` method defined in Code 4.4.

4.6.3 Testing

Whenever an UI Pattern is detected, the corresponding Test Patterns are applied: verify if the *pre-condition* is met, execute the necessary *actions* and verify the *checks* to decide if the test passes or fails. If everything goes smoothly, *i.e.* if all *checks* are met, the test passes. Otherwise, the test fails. At the end a report is generated. This corresponds to the `apply_test_pattern()` method of Code 4.1. If no Pattern is found in the Pattern Matching phase, this phase is skipped.

Both applying an UI Pattern and applying a Test Pattern follow, thus, Code 4.4.

Code 4.4: Apply pattern: find_UI_patterns() and apply_test_pattern()

```
if pattern precondition holds then
    call execute pattern actions
    call verify pattern checks
    if checks are met then
        return true
    else
        return false
    endif
else
    return false
endif
```

In some cases, there is more than one Test Pattern associated to an UI Pattern. In these cases, they are applied in sequence. This sequence is defined when the pattern is implemented. However, this could be improved in order to enable the user to decide the order in which the Test Patterns should be applied.

After this phase, the exploration phase is resumed.

If a pattern (either UI or Test) contains at least an action that is not just an *observation*, a *goBack()* method must be defined. This method attempts to return to a previous state of the application.

4.7 Implementation details

This Section presents some details of iMPAcT's implementation.

Installing the AUT and iMPAcT

iMPAcT's configurator is the one responsible for installing both the AUT, when needed, and the iMPAcT tester. This is performed with the `adb install -r <name_of_file>.apk` command, which is run through the `Runtime.getRuntime().exec()` method.

The command is run twice: one for installing the AUT's, using the name of the AUT's APK as the `<name_of_file>` and second one to install iMPAcT's tester, by replacing `<name_of_file>` by `iMPAcT`.

Finally, iMPAcT's configurator builds the configuration file with all the settings provided by the user and the package name of the AUT and copies it into the device.

Obtaining the AUT's package name

When the user provides the APK of the AUT, it is necessary to obtain its package name in order to, afterwards, start the application. iMPAcT's configurator achieves this by running the command *aapt d badging <apk_full_path>*. This returns the information present on the AUT's manifest file, including its package name.

List of application's permissions

One of the main aspects of the approach implemented in the iMPAcT tool is that it does not require access to the source code, which includes the manifest file where the permissions are defined. Thus, there are two possible alternatives to obtain the list of permissions requested by the AUT. The first one is to read them from the information returned from the *aapt* command previously mentioned in this Section. However, this requires the user to provide the APK of the application. The second alternative, which is the one followed, accesses this information during the execution of iMPAcT's tester through the method *PackageManager.getInstalledPackages(PackageManager.GET_PERMISSIONS)*. This returns all the permissions requested by all the applications running on the device, including the AUT.

Starting the Application

Before entering the exploration/testing cycle, it is necessary to start the AUT on the device. This is divided in four steps:

1. turn needed resources on;
2. set the catalogue;
3. set the device ready;
4. start the AUT.

Knowing which are the resources whose access is requested by the AUT, by accessing the corresponding service manager (*e.g.*, *WifiManager*), it is possible to turn them on. It is important to state that if iMPAcT does not request the corresponding permissions in its own manifest it will not be able to access these services.

Setting the catalogue implies reading the configuration file and adding an entry in the catalogue for each pattern defined and for each configuration.

Setting the device ready amounts to ensuring it is awake (not in sleep mode) and it is unlocked. Even though it is possible to verify whether the device is awake, it is not

possible to verify if it is unlocked. If the device is asleep, waking it up ensures it is in a locked state. However, if it is awake it can either be locked or unlocked. Thus, if the device is awake, the first step consists in putting it to sleep. At this point, the device is in the same state regardless of its initial state: asleep. Afterwards, the screen is awakened and, then, unlocked by swiping the screen from bottom to top. The `UiDevice.isScreenOn()`, `UiDevice.wakeUp()` and `UiDevice.swipe()` are used. Moreover, the device's orientation is set to portrait with `UiDevice.setOrientationNatural()`. This is depicted in Code 4.5.

Code 4.5: Set device ready to interact

```
function setDeviceReady
  if screen is on then
    call sleep
  endif
  call wakeUp
  call unlock the screen
  if orientation is not natural
    set orientation to natural
  endif
```

At this point, iMPAcT ensured the device is awake, unlocked and in a portrait orientation. It may either be on the home screen or focusing a running application, though.

The package name of the home screen, which is necessary for the stop condition as explained later in this Section, varies with the device. Hence, the *click device's home button* event is executed to ensure there is no application currently open in the screen. Then, the home screen's package name is read.

Finally, the permissions requested by the AUT are read and the corresponding services are turned on and the application is started.

Exploration leaves the application

While exploring an application some events may open a different application (*e.g.*, clicking on a link, clicking on the *Share* button). This is detected by reading the package name property of any of the elements currently present on the screen and comparing it to the AUT's package name. Whenever the application changes, the device's *Back* button is pressed in order to resume the exploration.

There are two exceptions to this situation: when the home screen is reached (stop condition) or a crash is detected. How iMPAcT deals with these cases is explained next.

Stop Condition

When pressing the *Back* button results on the exploration getting to the home screen of the device, *i.e.* exiting the application, the exploration stops.

This situation is detected by verifying if the package name of the screen currently in display corresponds to the one of the home screen stored while setting the device ready for the exploration.

Application crashes

The iMPAcT tool considers the application crashed when the screen on display is a pop-up and its package name is “*android*”. This heuristic was determined by an empirical analysis of crashes. An example is depicted in Figure 4.7.

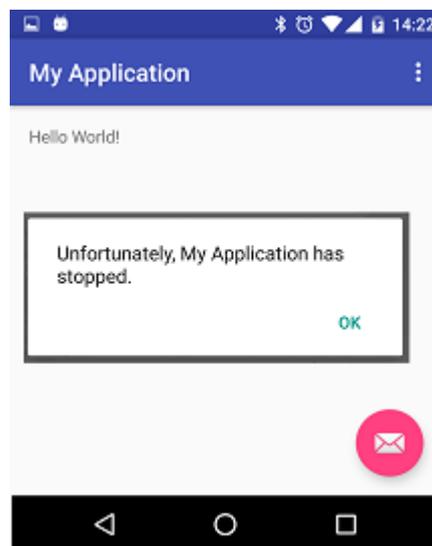


Figure 4.7: Example of a crash

A pop-up screen is defined as a screen whose root element does not occupy the whole screen.

Every crash screen contains an *Ok* button. When a crash is detected the exploration is resumed by clicking on it.

Comparing screens and activities

UIAutomator does not provide information about when the application reaches a new activity. So, two heuristics were defined: one implementing a basic level of comparison and another implementing a more complex level of comparison.

There is one aspect common to both heuristics: the name of the package of the screen. Each application has a package name associated to it that enables the device to identify the different applications that are installed. If two screens have elements with different package names then they do not belong to the same application and, thus, they do not correspond to the same activity.

First Heuristic: detect any difference

The first level of comparison consists in identifying if any change occurred in the screen and, thus, the most basic heuristic is applied: each of the elements are identified and compared and, if any element is present in only one of the screens, then a change on the screen is detected.

This heuristic is used when:

Deciding when to apply the patterns: if after executing an event no change is detected in the screen, then there is no need to spend time applying the patterns;

Verifying if no change occurred before executing the event: sometimes screens take some time to load all their contents. Therefore, occasionally the screens suffers modifications from the time it is read to the execution of the event. As such, before an event is executed, the current screen must be compared to the one analysed at the time of the decision to detect if any change occurred. If so the decision process is repeated. Only when no change is detected is the chosen event executed;

Verifying if scrolling the screen actually scrolled: UIAutomator indicates whether or not an element is scrollable. However, it does not provide any means of knowing the direction in which an element may be scrolled nor any information of whether it is still possible to scroll an element in a certain direction (Figure 4.8 depicts a scrollable list that is currently at its end: scrolling down the list will not change anything on the screen).

Verifying if opening the menu is a valid event: Devices that were prepared for versions prior to Android 4 presented a menu button that when pressed opened a context menu. Even though devices prepared for Android 4+ do not provide this button as the guidelines indicate this button should be part of the application and should be of the exclusive responsibility of the developer, some older applications still use it. In order to ensure the full functionality of these applications in modern devices, Android still presents a menu button when it is necessary (see Figure 3.7). Even though UIAutomator provides a way of interacting with this menu, it does not indicate whether or not it is present. As such, after trying to interact with it, the screens are compared in order to verify if executing the event resulted in the opening of a menu.

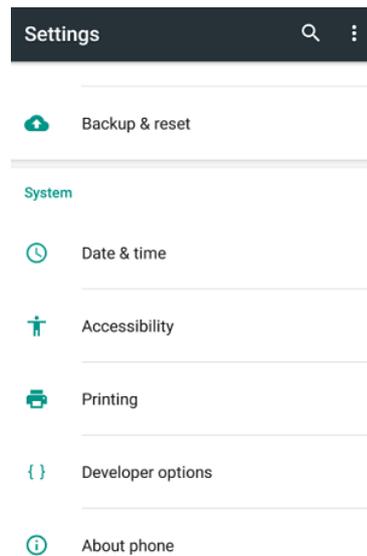


Figure 4.8: Example of a list that has been scrolled but has already reached its end (from Android Settings)

Second Heuristic

The second heuristic is used to differentiate activities, *i.e.* to detect if a change on a screen represents a different activity or if it is the same activity presenting some differences. This heuristic consists of four aspects.

The first aspect to be considered (after comparing the package names) is the dimensions of the root element of each screen: a screen that occupies the whole screen (full screen) is considered different than one that occupies the whole screen except for the notification bar and both are considered different than one that occupies just part of the screen (pop-up).

Secondly, the presence of a side drawer is detected. If only one of the screens contains a side drawer, then they are considered different. The presence of a side drawer is detected as explained in Section 4.8.1.

Thirdly, the elements to be compared are identified. Only the widgets are considered at this phase. If the screens present a side drawer only the widgets belonging to them are considered. On the other hand, if none of the screens contains a side drawer then all the widgets of the screen are considered except for the ones belonging to the action bar (see Figure 3.3).

Finally, the comparison of the widgets from each screen takes place: considering the previously identified activity is represented by screen *A* and the new screen is represented by screen *B*, then screen *B* represents a new activity if and only if none of the widgets

considered in screen *A* are present on screen *B* (only the widgets selected in the previous step are considered).

The comparison of the widgets disregards the modifiable properties of the widgets, *i.e.* it uses the second heuristic for elements comparison that is explained next. For example, a check-box is not considered different if in one of the screens it is checked and in the other it is unchecked.

Moreover, in order to avoid considering two screens different just because the position of the widgets changed (*e.g.*, some elements have a contextual menu associated to them, *i.e.* a pop-up with actions such as *edit* or *delete* that act on the corresponding element; two elements in different positions may open the same menu in different positions as depicted in the example of Figure 4.9), a final verification takes place in which both the modifiable properties and the position are disregarded. However, in this case they represent the same activity if and only if all the elements of screen *B* have an equivalent in screen *A*.

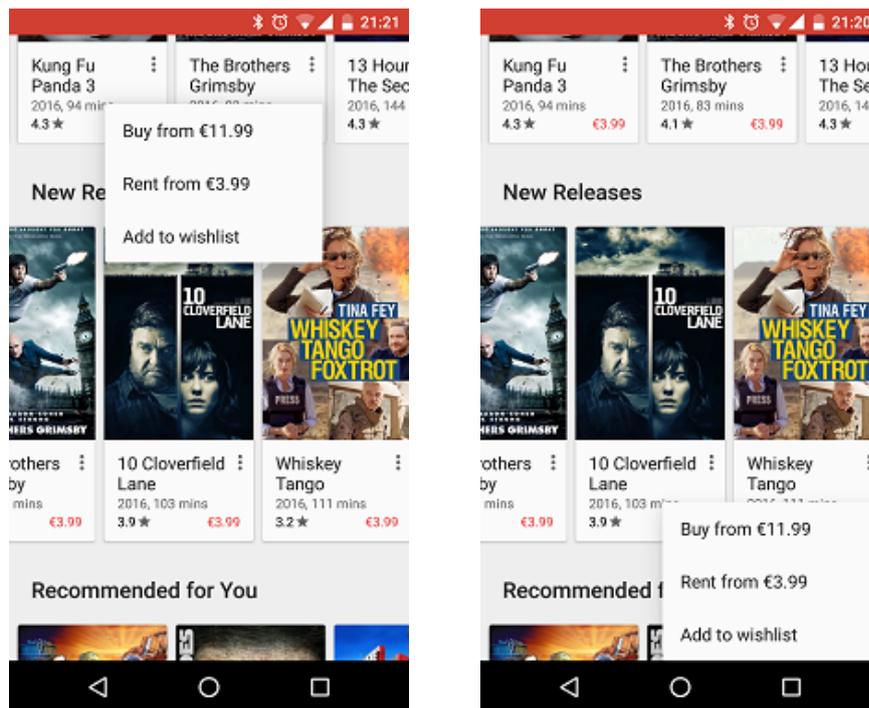


Figure 4.9: The same contextual menu displayed in two different positions

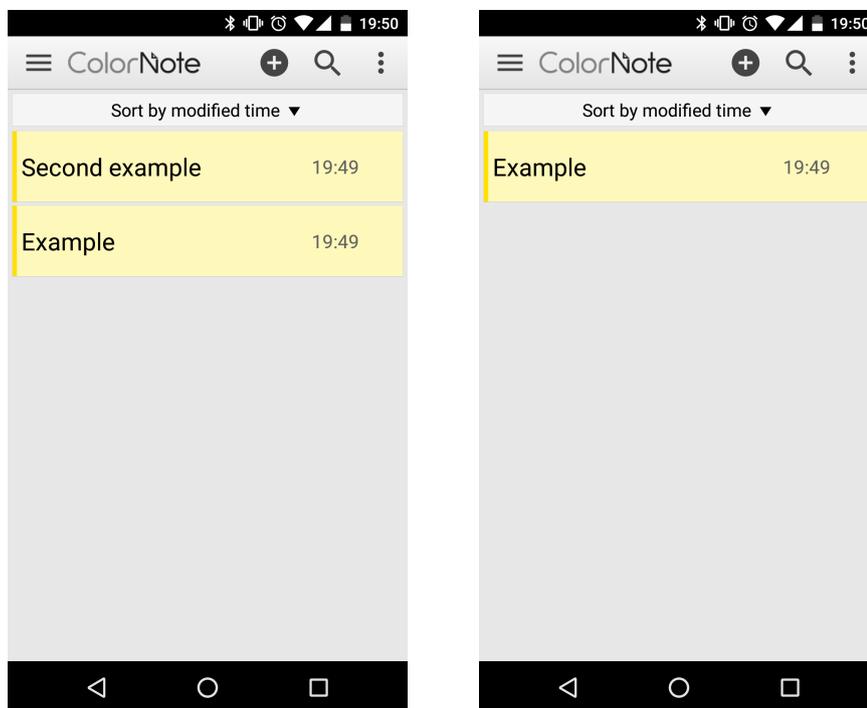
In summary, two screens represent the same activity if all the following aspects are verified:

- their elements have the same package name;
- their root elements have the same dimensions;

- either both or none contain a side drawer (if both screens contain a side drawer only the widgets belonging to it are considered);
- they have (the widgets that belong to an action bar are not considered):
 - at least one widget in common disregarding their modifiable properties OR
 - they have all widgets in common disregarding their modifiable properties and position

Comparison of elements

It is usually easier to automatically conclude that two elements are different than to conclude that they are the same. For instance, if the elements are of different types (*e.g.*, a button and a text) they are automatically considered different. However, to conclude, for example, that two edit texts are different it is not enough to compare all their properties as a change of the text does not mean the elements are different. Additionally, if a list item changes position because the one before it was removed from the list, the list item must also be identified as being the same as before. The example in Figure 4.10 depicts this situation: “Example” is the same in both screens even though its position changed.



(a) Before removing “Second Example”

(b) After removing “Second Example”

Figure 4.10: Element “Example” is the same regardless of its position

In order to solve this problem four heuristics were defined to compare two elements:

- The first and simpler one considers that an element is equal to another element only if all their properties are the same. This is only used for verifying if an element that has already been considered the same has suffered any alteration;
- The second heuristic compares two elements regardless of their modifiable properties: text, checked, selected, focused, clickable and scrolled. However, the text property may be used to aid the comparison. For instance, if both elements are textViews and they have the same non-empty text and both their parents are the same, the elements are automatically considered the same. This is used, for instance, when verifying if the user inserted data on any element;
- The third heuristic compares two elements disregarding their position but considering their modifiable properties. This is mostly used when testing the Orientation Pattern as the position (and sometimes even the dimensions) of an element changes when the screen is rotated;
- Finally, a fourth heuristic was defined to bring together both the second and third heuristics: disregard both the modifiable properties and the position of the elements being compared. This is used, for instance, when verifying if the user input data disappears when applying the Orientation Test Pattern.

Taking into consideration the type of information that may be provided about the different elements, these heuristics are capable of correctly comparing most elements.

However, developers seldom provide all the necessary information. This arises problems when comparing check boxes, radio buttons or switches as they usually have no text nor content description nor label associated to them. As such, an heuristic that complements all the previous ones (including the one that compares all the properties) was defined when comparing these types of elements.

This heuristic is based on the identification of the *identification text* of the element. The text, content description and label properties of the element under consideration are identified. If one of these properties is defined, it is considered the *identification text*. If more than one is defined the order in which they are considered the *identification text* is: 1) label, 2) content description and 3) text.

If none of these properties is defined, then the text next to the element must be identified and assumed as being the *identification text*:

1. identify the closest ascendant that has a descendant that is a text view;

2. identify all the text views that are descendant of that ascendant (it may be more than one as depicted in Figure 4.11).

The texts of all these *Text Views* are considered *identification texts* of the element.

Finally, the elements under comparison are considered the same if and only if at least one of their possible labels match.

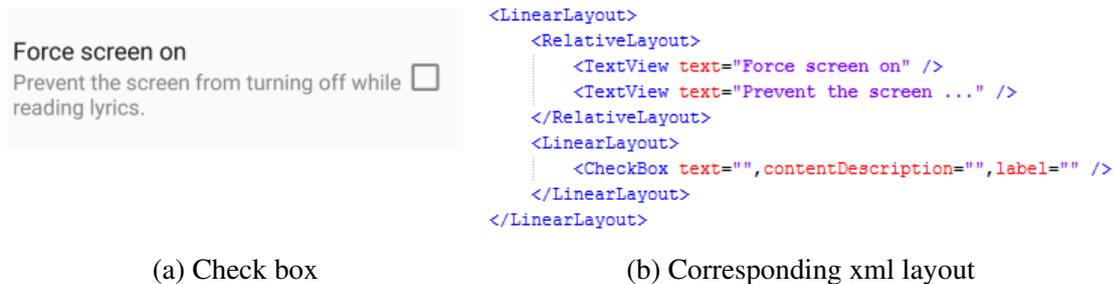


Figure 4.11: Check box and corresponding layout in the QuickLyric application

Comparison of events

Comparing events consists in comparing the respective names, input values and elements. The elements are compared using the heuristics previously defined. It is initially done using the first heuristic. If no match is found, it applies the second heuristic. If the screen was scrolled then the third and fourth heuristics (of elements comparison) are applied (instead of the first and second, respectively).

Information Stored During Execution

In order to implement the different exploration algorithms and the different patterns it is necessary to store some information during the exploration.

One aspect that is common to all exploration algorithms and to the patterns is the necessity of detecting if a screen has already been visited and, if so, which of its events have already been executed. Thus, a list of the different screens visited throughout the exploration is stored, *i.e.* iMPAcT detects if it is the first time visiting a certain screen, *i.e.* it is a new activity, and, if so, it then adds it to the list of visited activities; otherwise, iMPAcT updates the activity previously recorded in order to include the possible differences. This update is explained in Code 4.6.

Code 4.6: Update Activity

```
for each leave on new screen
  if leave present in previously saved activity
    update leave properties on previously saved activity;
  else
    find first ascendant to be present on the previously
      saved activity
    add the new branch to found element
  endif
```

In summary, iMPAcT stores information of the different activities that were explored throughout the execution and each of these activities include all the elements that have been part of that activity at some point of the exploration. The properties are always updated to their most recent values.

Each activity includes a list of the events that have already been identified and a list of the ones that have already been executed. Moreover, each activity also includes information on whether or not each of its scrollable elements can still be scrolled.

Finally, each event indicates the screen to which it leads. This value is only instantiated after the event has been executed.

Reduction of the number of events

The events associated to an element depend on the properties of that element. For instance, if an element is clickable then the *click* event is added. However, there are two main situations in which events do not need to be considered enabling a reduction of the number of executed events and, thus, of the overall duration of the exploration.

The simpler situation is when two events represent the same action. For instance, a check box is often both clickable and checkable even though the result of the event is the same. In these cases, only one of the events is considered.

The second situation occurs when an element is part of another element and both elements have the same actionable property (both clickable, for instance). Figure 3.1 in Section 3.1 depicts a layout (3.1a), with the corresponding *xml* structure (3.1b), where this is very common: a list of elements in which both the list items and the list itself are clickable.

Even though it may seem illogical, sometimes the *ListView* is incorrectly implemented as clickable (only its items should be clickable). In order to avoid this, an event is only considered if the element has no elements within its boundaries on which the same event

can be fired. This means that in the example of Figure 3.1 the event “*click on the list*” would be dismissed.

Detecting and closing the keyboard

The input method for textual elements in a smartphone is a keyboard that pops up in the screen. This keyboard appears whenever focus is given to an edit text box, which is the element in which text can be inserted. This keyboard is usually a set of touch screen keys that appear from the bottom of the screen and that, when pressed, modify the *text* property of the *edit text* element.

As this set of keys partially covers the screen, several elements become inaccessible when the keyboard is opened. However, UiAutomation is not capable of reading this layer, *i.e.* it considers that the elements hidden by the keyboard are still present and that it is still possible to interact with them. This raises several problems:

1. UIAutomator is capable of identifying the hidden element but the attempt of interacting with the element will result in an incorrect outcome as the keyboard is on top of the element;
2. UIAutomator is not capable of identifying the hidden element (no *UiObject2* is identified) and the event execution fails;
3. If the keyboard is opened when rotating the screen, it usually takes most of the screen, which may result in several different outcomes:
 - the elements behind the keyboard are not identified provoking an incorrect failure detection;
 - it is harder to scroll the screen as the available section to correctly scroll is too small;
 - the *UiScrollable* is not identified and the scroll fails;
 - the scroll is performed on the keyboard provoking an unintentional insertion of text in the focused edit text.

Considering all these problems, the best solution is to close the keyboard whenever it is opened. In order to close the keyboard the *click device's back button* event is executed. However, if this event is executed when the keyboard is not opened, the exploration will be affected. Thus, it is important to correctly identify when the keyboard is opened, which is a challenge as neither UiAutomation nor UIAutomator are capable of detecting its presence.

The solution for this problem is depicted in Code 4.7 and consists in identifying if there is an *AccessibilityWindowInfo* of the type *AccessibilityWindowInfo.TYPE_INPUT_METHOD* present on the screen.

Code 4.7: Code for identifying if a keyboard is open on the screen

```
function isKeyboardOpen
    for each window in UiAutomation.getWindows
        if window.type is TYPE_INPUT_METHOD then
            return true;
        endif
    endfor
    return false
```

Therefore, whenever a keyboard is detected, the *back* event is executed, thus closing it, regardless of the state of the exploration or the state of testing.

Identifying the Action Bar and the App Button

The identification of the action bar follows an heuristic that, alike the one defined for identifying the side drawer, was based on the empirical analysis of several applications.

An action bar is detected when an element (the root element of the action bar) with one of the following characteristics is present:

- it is of the type *frameLayout* and
 - its *resource_id* is “*action_bar_container*” and it has a child that is either a view with “*action_bar*” as its *resource_id* or an *horizontalScrollView* with “*action_bar_tab_scroller*” as its *resource_id*, OR
 - it has a child that is a view and whose *resource_id* is “*toolbar*” and its *resource_id* is “*action_bar_placeholder*” or “*toolbar_container*” or “*action_bar_placeholder*” or is undefined.
- it is a view whose *resource_id* is “*toolbar*”;
- it is a *viewGroup* whose *resource_id* is either “*action_bar*” or “*toolbar*”.

The App Button can only be identified if an Action Bar is detected. In this case, the clickable descendants nearest to the left edge of the Action Bar is identified. If the Action Bar does not contain any non-clickable elements of the type *ImageButton* or *ImageView* to the left of the element, then it is considered to be the App Button. Otherwise, the Action Bar does not contain an App Button.

Creation of the UI Model

The UI Model obtained at the end of the exploration consists, as explained in Section 3.7, in a FSM in which the states are the different activities of the application and the edges are the events that is necessary to execute to go from the origin state to the destiny state.

The creation of the UI Model is performed in a post-exploration phase and involves two steps:

1. at the end of the exploration/testing process iMPAcT's tester builds a JSON⁶ with the information about the screens and events and saves it into a file;
2. iMPAcT's configurator reads the file and transforms it into a visual graph using the `mxGraph`⁷ and Java Swing⁸ classes;

After creating the graph, its layout can be manually rearranged.

When the user selected the option to take screenshots of the activities, iMPAcT's tester uses `UiDevice.takeScreenshot()` method to take a screenshot of the screen whenever a new activity is detected and, at the end, iMPAcT's tester obtains those files from the device and uses them to represent the different states. Otherwise, the states are represented by ovals. The *crash* and *out of AUT screens* are represented by ovals regardless of the user's preference. An example of a UI Model can be depicted in Figure 5.5 in Chapter 5.

4.8 Patterns Catalogue

One of the main components of iMPAcT is the Patterns Catalogue. At the moment four patterns are implemented: Side Drawer, Orientation, Resource Dependency and Tab. The implementation of these patterns presented some challenges. This Section describes how these challenges were overcome and other details on the patterns implementation.

4.8.1 Side Drawer Pattern

In order to identify and test this pattern, it is necessary to:

- open the side drawer;
- identify if a side drawer is present on the screen;
- compare the height of the side drawer element to the height of the screen.

⁶<http://www.json.org/>

⁷<https://jgraph.github.io/mxgraph/>

⁸[https://en.wikipedia.org/wiki/Swing_\(Java\)](https://en.wikipedia.org/wiki/Swing_(Java))

Even though the definition of the UI Pattern states that it just verifies if there is a hidden side drawer, UIAutomator does not allow this verification. Thus, the “*open side drawer*” event must be part of the UI Pattern in order for it to verify its existence (UI Pattern’s check) and the Test Pattern only verifies its dimensions. Before executing the “*open side drawer*” event, the UI Pattern verifies if it is already open and, if so, it just applies the Test Pattern.

Open the side Drawer

There are two ways of opening a side drawer: swiping the screen or clicking the app button. However, sometimes swiping the screen fails to open it and sometimes there is no app button even though there still is a side drawer available. Thus, in order to open the side drawer, iMPAcT first swipes the screen from the left edge to the right edge (using *UiDevice.swipe()*) and verifies if a side drawer was opened. If it was not, iMPAcT identifies the app button (as described in 4.7) and, if it is present, clicks on it. In order to avoid clicking the *up button* instead of the *app button*, iMPAcT only clicks it if neither its *text*, *content description* nor *resource_id* contain the word *up*.

Identify if a side drawer is present on the screen

This was the most challenging aspect of implementing this pattern and it was achieved by following an heuristic that identifies an element as the root of the side drawer when it has all of the following characteristics:

- it does not take up the whole screen;
- it starts on the left edge of the screen;
- it does not take up the full width of the screen;
- it takes up at least half of the height of the screen;
- it is not the root of the screen
- it sits on top of other elements, *i.e.* there are items behind it;
- the hierarchy of elements follow one of these aspects:
 - a) the element is an only child and it is of the type *listView* or *frameLayout* and its parent is a *View* and its grandparent is a *frameLayout*;
 - b) the element is not an only child and it is either a *frameLayout* or a *linearLayout*.

Moreover, if the screen contains a pop-up the possibility of a side drawer is automatically dismissed.

Compare the height of the side drawer element to the height of the screen

Once the element that is the root of the side drawer is identified, it is only necessary to get its position on the screen and compare its y coordinates (*AccessibilityNodeInfo.getBoundsInScreen()*) with the dimensions of the device (*UiDevice.getDisplayHeight()*).

Restoring previous state (*goBack()*)

In the case of this pattern, the *goBack()* method consists on closing the side drawer whenever it has been opened. This is achieved by injecting a click event in any part of the screen that does not coincide with the side drawer. This successfully returns to the previous state of the application.

4.8.2 Orientation Pattern

In order to identify and test this pattern, it is necessary to:

- verify if the user inserted new data;
- rotate the screen;
- match the elements of the before screen to the ones of the after screen;
- compare the modifiable properties of the elements that received user input.

Verify if the user inserted new data

In order to do this, it is only necessary to access the modifiable properties of the elements (text, checked and selected) and compare them to those elements' properties before the event was fired. This match uses the second heuristic of the comparison of elements presented in Section 4.7.

Rotate the Screen

The *UiDevice* class contains three methods to handle the rotation, depending on the direction: *setOrientationLeft()* that rotates the screen to the left, *setOrientationRight()* that rotates the screen to the right and *setOrientationNatural()* that rotates the screen until it

is on the AUT's natural orientation. Plus, it is possible to verify the current orientation of the device with *getDisplayRotation()*.

Even though the UI Pattern definition states that it only verifies if it is possible to rotate the screen, UIAutomator is not able to verify this. Thus, the implementation of the UI Pattern only verifies if the Orientation Pattern has not yet been applied to current activity. After trying to rotate the screen, iMPAcT verifies if the screen's orientation changed. In case it has remained unaltered (the orientation did not change), the Test Pattern passes while logging its incapability to rotate the device.

Match the elements of the before screen to the ones on the after screen

In Section 4.7, the different heuristics to compare elements were defined. In order to match the elements the third heuristic was used, *i.e.* the elements are compared without considering their position.

When rotating the screen the layout of the screen changes which often provokes modifications on the structural elements, which may include their disappearance. This, however, does not constitute a failure. Therefore, only the widgets are matched.

Compare the modifiable properties of the elements that received user input

The second Test Pattern associated to this UI Pattern verifies if user input data did not suffer any modification when rotating the screen. Thus, it is necessary to match the element that received user input of the before screen to the one on the after screen. In order to do this, the fourth elements comparison heuristic (Section 4.7) is used, *i.e.* nor the position nor the modifiable properties of the elements are considered when comparing the elements.

Restoring previous state (*goBack()*)

In this pattern, the *goBack()* method consists in rotating the screen back to the initial orientation. This does not always successfully return to the previous state of the application. For instance, if a dialogue disappears when the screen is rotated (example in Figure 5.1), it will not be present when the screen is rotated back to the initial orientation.

4.8.3 Resource Dependency Pattern

In order to implement this pattern it is necessary to:

- verify if the resource is used;

- read and modify the resource status.

As it is possible to access the Context of the Instrumentation package, it is also possible to access all the service's managers of the device (*Context.getSystemService()*).

Verify if the resource is used

UIAutomator is not capable of verifying if the resource is currently in use. Thus, the best solution was to verify if it is possibly using it, *i.e.* if it requested permission to use it. This is explained in Section 4.7. If a permission for the resource in question was requested (*e.g.*, *android.permission.INTERNET* is the permission for internet access) then the resource is considered to be in use.

Read and modify the resource status

In order to read and modify the status of a given resource, iMPAcT accesses the manager of the corresponding service (*e.g.*, the *WifiManager*), verifies its status (*e.g.*, *WifiManager.isWifiEnabled()*) and modifies it (*e.g.*, *WifiManager.setWifiEnabled(false)* turns off the Wifi).

Restoring previous state (*goBack()*)

In the context of this pattern going back consists in turning the resource back on. This is effective if no failure is detected. Otherwise, the “OK” button of the crash screen is clicked. However, this does not ensure that the application returns to the original state.

4.8.4 Tab Pattern

In order to implement this pattern it is necessary to:

- verify the presence and quantity of the sets of tabs;
- verify the position of the set of tabs;
- horizontally swipe the screen to change the selected tab;
- verify which tab is selected.

Verify the presence and quantity of the sets of tabs

There are two classes of elements that can be the root of a set of tabs: *horizontalScrollView* and *tabWidget*. In order to detect the presence of tabs in a screen at least one element of one of these classes must be present on the screen. The quantity of sets of tabs is defined by the quantity of elements of any of these classes.

Verify the position of the set of tabs

In order to verify the position of the tabs it is only necessary to access the coordinates of the element representing the root of the set of tabs.

The guidelines define the tabs should be on the upper part of the screen but no indication of what should be considered the upper part is provided. As such, dividing the screen into three equal parts, if the set of tabs is positioned on the first of these parts, then it is considered to be on the upper part of the screen, *i.e.* it is correctly positioned.

Horizontally swipe the screen to change the selected tab

In order to swipe the screen, the *UiDevice.swipe()* method is used. By default, the screen is swiped left, *i.e.* the new tab selected will be the one on the right of the previously selected tab. However, if the tab that was selected before the test is applied is identified as being the last tab, the screen is swiped left, *i.e.* the new tab selected will be the one on the left of the previously selected tab.

Verify which tab is selected

This step should be as simple as to read the *selected* property of each of the tabs present on the screen. When this happens, it is only necessary to verify if the selected tab changed after swiping the screen. However, occasionally developers do not update this property after selecting a tab, resulting in zero or multiple tabs with the *selected* property set to *true*. When this is the case it is not possible to identify the selected tab in a simple manner. In this situation, the content of the screen below the set of tabs before swiping the screen is compared to the content of the same screen section after swiping the screen. If the content changed the Test Pattern passes and the screen is swiped in the opposite direction. Otherwise, it fails. Here, the first heuristic for the elements comparison is used.

Restoring previous state (*goBack()*)

In the context of this pattern, the *goBack()* method is only applied if the *Horizontally scrolling the screen should change the selected tab* test pattern is applied and if the se-

lected tab changed. In this case the screen is swiped in the opposite direction. This method is always effective in returning to the previous state of the application.

4.9 Conclusions

This Chapter presented the implementation details of the iMPAcT (Mobile Pattern Testing) tool, in which the approach presented in Chapter 3 was implemented.

The iMPAcT tool was written in Java and uses the UI Automation API to read the contents of the screen and the UIAutomator framework to simulate the interactions. Both UI Automation and UIAutomator are available on the Android SDK. The tool only requires the identification of the AUT (either by providing the APK or the package name) and it produces a report, which includes the log of the exploration and whether or not the patterns are detected and correctly implemented, and a FSM with the different screens of the AUT and the events that provoke a change of screen.

The main contributions of this tool are:

- it automatically explores the GUI of an application;
- it provides three algorithms to guide the exploration of the application;
- it is capable of automatically detecting the presence of UI Patterns;
- it is capable of automatically testing the UI Patterns that were detected;
- it produces a report on the tested UI Patterns and a FSM and an exploration log representing the exploration.

In spite of the advantages iMPAcT provides when automating mobile testing, it still presents some limitations:

1. there are some events that are not yet implemented, like zooming in and out of a view;
2. some system events are not yet possible to fire (*e.g.*, simulating a phone call);
3. the exploration process could be improved by offering, for instance, a self-learning algorithm;
4. its usefulness in terms of testing depends on the quantity of patterns present in the AUT.

Nevertheless, some of its limitations are due to developers not providing all the accessibility information they should. For instance, it is only possible to detect if an option is selected if the option *selected* is set to *true*. In some cases, the developers visually select an element (changing its background colour, for example) but the *selected* property's value remains *false*.

UIAutomator and UiAutomation also present some issues that limit the capabilities of iMPAcT:

- there are some attributes missing from the elements' characterisation. For instance, there is no attribute indicating the (background) colour of an element. This type of properties could be used in the definition of new heuristics to identify the selected tab, by analysing the visual differences between the tabs before and after swiping the screen in the Tab Pattern;
- the event's execution has some issues, which are out of iMPAcT's control (*e.g.*, on one of the explorations iMPAcT fired the event "*scroll spinner*", which was inside a wider scrollable view and the UIAutomator scrolled the involving view). This is similar to a problem that is commonly known in web applications as *double scroll*;
- some elements are not read by the UiAutomation, such as toasts⁹ or the incoming call pop-up.

Despite these limitations, UIAutomator is still the framework that best suits the requirements for this implementation.

⁹<http://developer.android.com/guide/topics/ui/notifiers/toasts.html>

Chapter 5

Experiments

One of the main aspects of any research is the validation of the hypothesis and corresponding approach making it of high importance to carefully define the appropriate research questions and to design experiments that respond to them. This Chapter presents the six research questions defined and presents the three experiments designed to answer them.

This Chapter is structured as follows. Section 5.1 presents the research questions. Section 5.2 presents the technical specifications of the device on which the experiments were conducted. Sections 5.3, 5.4 and 5.5 present the three designed experiments and corresponding results. Section 5.6 presents an example of the artefacts produced by iMPAcT in one of its executions. Section 5.7 presents the drawn conclusions. Chapter 6 presents the discussion of the results presented along this Chapter.

5.1 Research Questions

In order to validate the research hypothesis presented in Chapter 1 and the corresponding approach presented in Chapter 3, the following research questions (RQ) were defined:

RQ1 Is iMPAcT able to identify failures in Android Mobile Applications?

RQ2 Is iMPAcT's performance affected by the size of the applications? This can be divided into two sub-questions:

- a) is the coverage acceptable, *i.e.* is it capable of executing the majority of the events it identifies?
- b) does the whole process take a reasonable amount of time?

RQ3 Does the exploration algorithm used affect the results, performance and coverage of iMPAcT?

RQ4 Does the order in which the patterns are identified and tested affect the results, performance and coverage of iMPAcT?

RQ5 Are the results provided by iMPAcT sensitive, *i.e.* are the existing failures properly identified by iMPAcT?

RQ6 Does iMPAcT reliably identify failures, *i.e.* are the failures detected by iMPAcT actually failures?

The remaining of this Chapter presents the experiments defined to successfully answer these questions.

5.2 Technical Specifications

For all the experiments described in this Chapter, iMPAcT was run on a LG Nexus 5, which presents the following main characteristics:

- Operating System: Android 6.0
- CPU: Quad-core 2.3 GHz Krait 400
- Chipset: Qualcomm MSM8974 Snapdragon 800
- GPU: Adreno 300
- RAM: 2GB

The characteristics of the computer used are not relevant because they do not affect neither the performance nor the results as iMPAcT is solely run on the device making use of none of the computing capacity of the computer.

5.3 Detection of UI failures

The objective of this experiment presented in this Section is to validate the feasibility and effectiveness of the approach presented in this document, *i.e.* it was designed to answer research questions 1 (*is iMPAcT able to identify failures in Android Mobile Applications?*) and 2 (*is iMPAcT's performance affected by the size of the applications?*). This experiment was peer-reviewed and published in [Coimbra Morgado & Paiva, 2015a].

For this experiment the four patterns presented in Section 3.3 were considered: Side Drawer, Orientation, Resource Dependency and TabScroll.

Even though iMPAcT enables the user to select the exploration algorithm, in this experiment, the more complex one, *priority to not executed and list items*, is the only one considered.

In summary, this experiment consists in running the iMPAcT tool on five applications that were either used by other researchers during their validation process [Amalfitano et al., 2014; Costa et al., 2014; Yang et al., 2013] or are official Google applications. Table 5.1 presents some characteristics of each of these applications.

Table 5.1: Applications tested with the iMPAcT tool

Applications	Size (MB)	Thousands of Installations	Domain	Reviews Classification	Number of Reviews
Book Catalogue ¹	9.81	100-500	Productivity	4.4	3121
Tomdroid ²	1.0	10-50	Productivity	3.9	737
Tippy Tipper ³	3.12	100-150	Finance	4.6	792
Google Slides ⁴	83.74	10k-50k	Productivity	4.1	117,809
Google Calendar ⁵	33.25	100k-500k	Productivity	4.1	484,095

The randomness associated to the decision of which event to fire affects the path taken during the exploration. For example, if an item in a list is removed prior to being explored the results obtained may be different than the ones obtained if the item is explored before being eliminated. Figure 4.10 in Section 4.7 depicts an example of this situation as if the “Second Example” is removed before iMPAcT interacts with it (Figure 4.10b) that path will not be explored. On the other hand, the number of events identified and executed, and consequently the percentage of events executed, depend on the path taken during the exploration in each run. Moreover, the total number of events that can be executed is not known beforehand making it necessary to identify (and distinguish) the events during the execution making the number of identified events also dependent on the path taken during the exploration. Therefore, for each application iMPAcT was run at least five times (reducing the effect of the randomness inherent to the choice) and the results presented in this Section are the average of the runs for each of them.

The information considered for this experiment consists in:

- *UI Pattern presence*: was each of the UI Patterns detected on each of the applications?

- *Failure detection*: in the cases where the UI Pattern was detected, did it present failures?
- *Execution time*: the time it took iMPAcT to completely explore each of the applications. This includes both the exploration and the testing processes as they are intertwined;
- *Events executed*: number of events executed by iMPAcT when exploring each of the applications;
- *Events identified*: number of events identified by iMPAcT: number of events identified by iMPAcT when exploring each of the applications.

The first two metrics illustrate whether or not iMPAcT is able of finding failures in the mobile applications, while the three latter provide information on the performance and coverage provided by iMPAcT.

At the time of the elaboration of this experiment the *press menu button* event was not implemented and the *scroll* event was only executed when testing the orientation pattern.

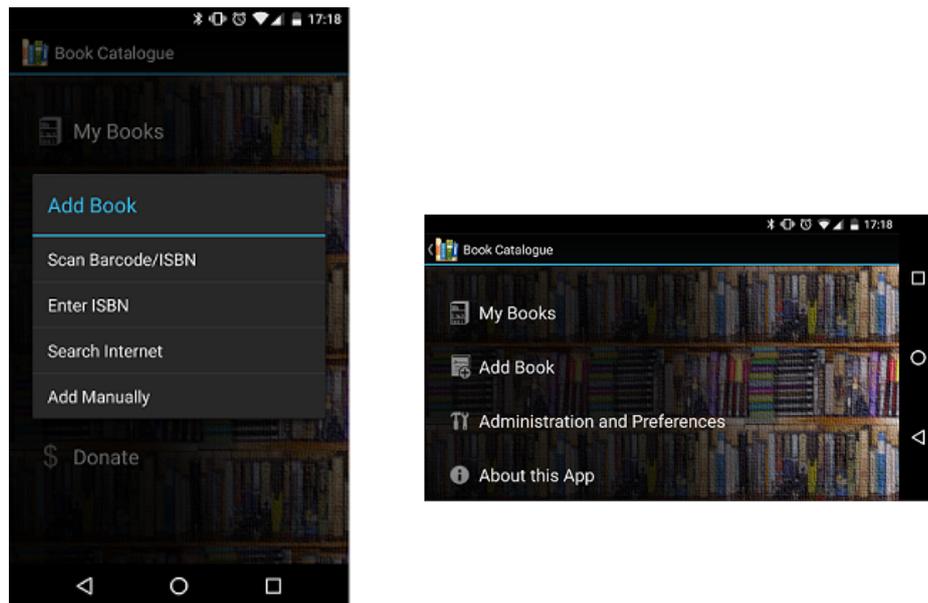
The results for the first two metrics are presented in Table 5.2.

Table 5.2: Summary of test results part 1: FF-Failure Found; AF-Absence of Failures; NA-Not Applicable

Applications	SideDrawer	Orientation	Resource (Wifi)	Tab
Book Catalogue	NA	FF	AF	FF
TomDroid	NA	AF	AF	NA
Tippy Tipper	NA	AF	AF	NA
Google Slides	AF	FF	AF	NA
Calendar	FF	FF	AF	NA

Analysing the report, which includes the traces and when a failure is detected, it is possible to exemplify some of the failures detected:

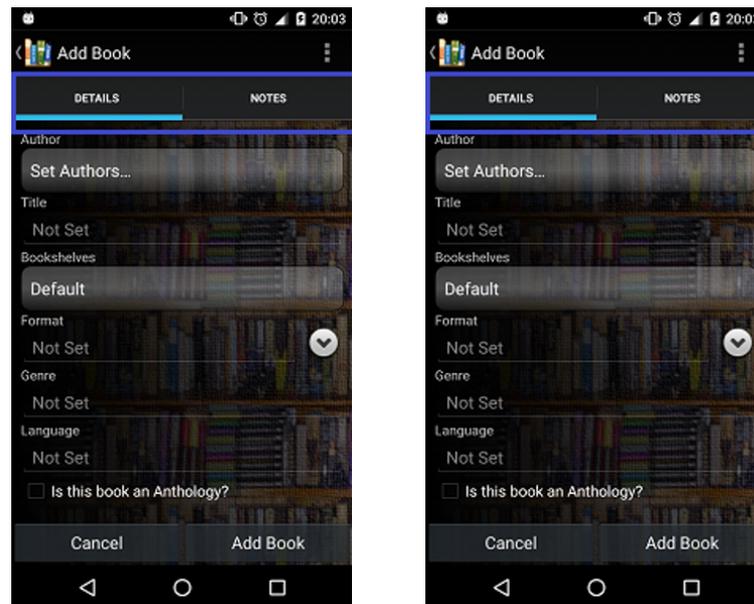
1. Book Catalogue: Orientation Pattern is not correctly handled. For example, when a pop-up appears, rotating the screen makes it disappear. An example is depicted in Figure 5.1;
2. Book Catalogue: Tab Pattern is not correctly handled. For example, when adding a book there are two tabs ("Details" and "Notes") but horizontally swiping the screen does not change the selected tab. This is depicted in Figure 5.2;



(a) Portrait: contains pop-up

(b) Landscape: pop-up disappeared

Figure 5.1: Rotating the screen makes the pop-up disappear



(a) Before Horizontal Swipe

(b) After Horizontal Swipe

Figure 5.2: Screen of Book Catalogue that contains the tabs: “Details” and “Notes”. Horizontally swiping the screen does not change the selected tab.

3. Tippy Tipper: Orientation Pattern is not correctly handled. After pressing the *Calculate* button (Screen 1 on the FSM of Figure 5.5), the *Round Down* button disappears. In fact, it is still present but its text is changed to *Down* and, thus, it is not identified as being the same button;

4. TomDroid: Orientation Pattern is not correctly handled. For example, when the “More Options” menu is open, rotating the screen makes one of the options (“Search”) disappear. This is depicted in Figure 5.3;

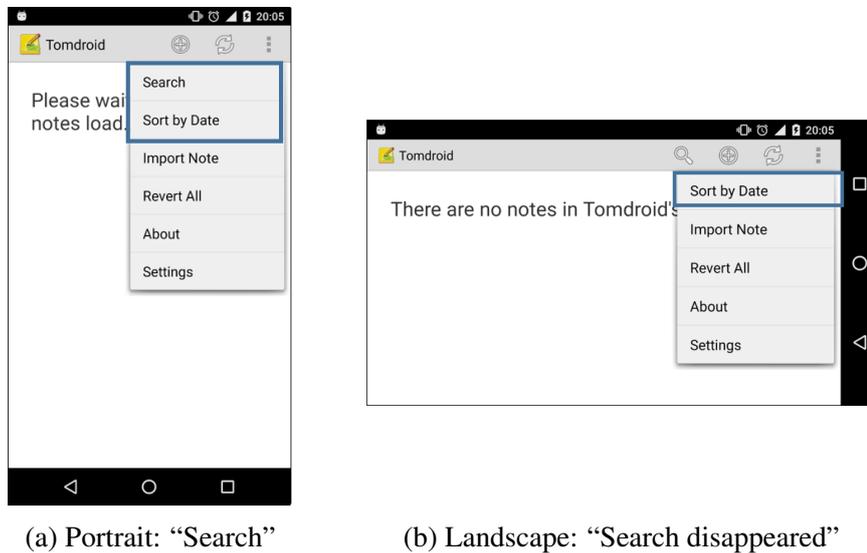


Figure 5.3: Rotating the screen makes the Search option disappear

5. Google Slides: Orientation Pattern is not correctly handled. For example, when opening the “More options” menu, rotating the screen makes this menu disappear;
6. Calendar: Side Drawer Pattern is not correctly handled: the side drawer does not take up the full height of the screen. This is depicted in Figure 5.4;

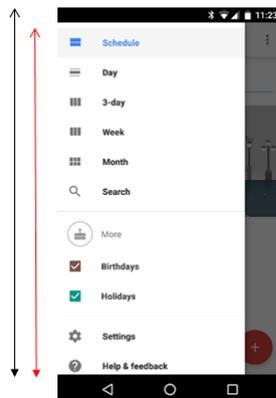


Figure 5.4: The Side Drawer of the Calendar application does not take up the full height of the screen

7. Calendar: Orientation Pattern is not correctly handled. For example, when selecting the month a calendar appears but rotating the screen makes it disappear; when open-

ing the *More Options* menu, rotating the screen makes it disappear; and rotating the screen at any moment makes the action bar disappear.

The results of the remaining metrics are presented in Table 5.3. These results represent the average of all the executions of each of the applications as previously explained. The *percentage of events* represent the percentage of events executed in regards to the number of events that were identified in each execution and the *percentage of events over all executions* represent the percentage of events executed in each execution in regards to the maximum number of events identified by all the executions of the same application. All these values are the average of the values obtained over the five executions.

Table 5.3: Summary of test results part 2

Applications	% of events	% of events over all executions	Execution Time
Book Catalogue	76	53	16m 49s 90ms
TomDroid	78	68	5m 4s 634ms
Tippy Tipper	93	74	4m 44s 659ms
Google Slides	69	55	19m 47s 512ms
Calendar	66	47	11m 0s 885ms

5.4 Impact of exploration algorithms and pattern identification order on finding failures

In the experiment presented in Section 5.3, only one of the exploration algorithms was used: the *priority to not executed and list items*. However, it is important to study how the choice of the exploration algorithm can affect the results produced by iMPAcT. Moreover, the order in which the patterns are identified and tested may also affect the results.

The objective of this experiment is to study how the iMPAcT tool's results are affected by changing the exploration algorithm and by the order in which the patterns are tested, *i.e.* to answer research questions 3 (*does the exploration algorithm affect the results, performance and coverage of iMPAcT?*) and 4 (*Does the order in which the patterns are identified and tested affect the results performance and coverage of iMPAcT?*). This experiment was peer-reviewed and published in [Coimbra Morgado & Paiva, 2016].

This experiment only considers the side drawer (SDP) and the orientation (OP) patterns as these are the ones that may considerably affect the exploration: in order to verify if the AUT presents a side drawer it is necessary to try to access it by swiping the screen

from its left edge to the right possibly causing the UI screen to change (for instance, if a menu or a pop-up screen is displayed, this action could cause it to close); if the orientation pattern is not correctly handled, rotating the screen may also disrupt the current state of the application (example in Figure 5.1).

In summary, this experiment consists in running the iMPAcT tool on two applications, Book Catalogue and TomDroid, with six different configurations: three exploration algorithms (*execute once*, *priority to not executed* and *priority to not executed and list items*) combined with two pattern testing orders (SDP followed by OP and OP followed by SDP).

Similarly to the experiment presented in Section 5.3, the coverage can only be calculated according to the number of events identified in the different executions of iMPAcT. Each configuration of iMPAcT was run on each of the applications several times to minimise the effect of the randomness associated to all the exploration modes. Thus, the results presented in this Section are the average of the set of runs with a certain configuration.

This experiment uses four concepts that must be understood:

- **execution:** running the iMPAcT tool on an application once;
- **exploration algorithm:** the algorithm used by the iMPAcT tool in a certain execution (1-execute once; 2-priority to not executed; 3-priority to not executed and list items);
- **testing order:** the order in which the patterns are identified and tested;
- **configuration:** a defined exploration algorithm and testing order.

The results obtained are divided in:

- **orientation failures:** average of failures detected in the orientation pattern (there is no column for the failures related to the side drawer pattern because there were no significant differences detected);
- **configuration time:** average of the time taken in each configuration;
- **events identified:** maximum number of events identified in the configuration;
- **% of events:** average percentage of executed events over events identified in each execution;

- **% of events over configuration:** the average percentage of executed events over the maximum number of events identified by executions of the same exploration algorithm and testing order;
- **% of events over exploration algorithm:** the average percentage of executed events over the maximum number of events identified by executions of the same exploration algorithm (regardless of testing order);
- **% of events over all executions:** the average percentage of executed events over the maximum number of events identified by all executions (regardless of testing order or exploration algorithm).

Tables 5.4 and 5.5 presents the results of the executions for the Tomdroid application.

Table 5.4: Results obtained when applying different configurations to Tomdroid - Part 1

Testing Order	Orientation failures	Execution Time
exploration algorithm 1: execute only once		
SD-O	1	2m 4s 410ms
O-SD	2.75	2m 8s 402ms
exploration algorithm 2: priority to not executed		
SD-O	1.75	5m 45s 466ms
O-SD	2.75	5m 50s 584ms
exploration algorithm 3: priority to not executed and list items		
SD-O	2	5m 39s 951ms
O-SD	2.5	5m 47s 346ms

Table 5.5: Results obtained when applying different configurations to Tomdroid - Part 2

Testing Order	Number of events identified	% of events	% events over configuration	% events over exploration algorithm	% events over all executions
exploration algorithm 1: execute only once					
SD-O	76	52.7	33.6	33.6	24
O-SD	51	67.6	54	36.2	26
exploration algorithm 2: priority to not executed					
SD-O	106	85.6	63	63	63
O-SD	89	76.3	75	63	63
exploration algorithm 3: priority to not executed and list items					
SD-O	89	82.2	74.5	65.6	62.5
O-SD	101	76.8	61.1	61.1	59.15

Table 5.6 and 5.7 present the results of the executions on the Book Catalogue application.

Table 5.6: Results obtained when applying the different exploration algorithms and pattern orders to Book Catalogue - Part 1

Testing Order	Orientation failures	Execution Time
exploration algorithm 1: execute only once		
SD-O	5.75	4m 27s 362ms
O-SD	9	5m 15s 798ms
exploration algorithm 2: priority to not executed		
SD-O	13.25	13m 24s 297ms
O-SD	12.75	8m 32s 318ms
exploration algorithm 3: priority to not executed and list items		
SD-O	23.75	34m 1s 170ms
O-SD	33.25	32m 13s 346ms

Table 5.7: Results obtained when applying the different exploration algorithms and pattern orders to Book Catalogue - part 2

Testing Order	Number of events identified	% of events	% events over configuration	% events over exploration algorithm	% events over all executions
exploration algorithm 1: execute only once					
SD-O	141	45.6	25	25	5.4
O-SD	141	50.5	23.3	23.3	6.5
exploration algorithm 2: priority to not executed					
SD-O	372	54	34.3	34.3	19.5
O-SD	230	52.7	34.4	20.8	12.1
exploration algorithm 3: priority to not executed and list items					
SD-O	653	79.1	52.2	52.2	52.2
O-SD	418	83.6	76.1	48.7	48.7

5.5 Quality of the tool results

The objective of this experiment is to study the sensitivity of the tool, *i.e.* if the failures present on the application are identified by iMPAcT, and its *PV+* rate, *i.e.* if the detected failures correspond to actual failures. This allows answering research questions 5 (*are the results provided by iMPAcT sensitive?*) and 6 (*does iMPAcT reliably identify failures?*).

This is achieved by analysing the true and false positives and the true and false negatives. This experiment was submitted for peer-review.

In the context of this experiment:

- a true positive occurs when a failure detected by iMPAcT is actually a failure;
- a false positive occurs when a failure detected by iMPAcT is not a failure;
- a false negative occurs when an existing failure is not detected by iMPAcT;
- a true negative occurs when iMPAcT correctly tests and does not identify a failure.

The validation of the detected/not detected failures is achieved by manually testing the applications. This may be summarised as in Table 5.8.

Table 5.8: Definition of positives and negatives

	iMPAcT		
Manual inspection		detects a failure	does not detect a failure
	is failure	true positive	false negative
	is not failure	false positive	true negative

From the gathering of the true/false positives/negatives, it is possible to calculate four metrics: sensitivity, specificity, predictive value for positive result and predictive value for negative result.

The sensitivity is calculated with Equation 5.1 and indicates the likelihood of a failure being identified.

$$sensitivity = \frac{truepositives}{truepositives + falsenegatives} \quad (5.1)$$

The specificity is calculated with Equation 5.2 and indicates the likelihood of a non-failure not being classified as a failure.

$$specificity = \frac{truenegatives}{truenegatives + falsepositives} \quad (5.2)$$

The predictive value for positive result ($PV+$) is calculated with Equation 5.3 and indicates the likelihood of a detected failure actually corresponding to a failure.

$$PV+ = \frac{truepositives}{truepositives + falsepositives} \quad (5.3)$$

The predictive value for negative result ($PV-$) is calculated with Equation 5.4 and indicates the likelihood of a non-detected failure actually not being a failure.

$$PV- = \frac{truenegatives}{truenegatives + falsenegatives} \quad (5.4)$$

However, the analysis of true negatives holds several concerns. As an example let us consider the orientation pattern. The pattern is only tested when on the presence of a new activity or when the user inserted new information. However, the detection of a new activity follows an heuristic and, thus, is susceptible to errors. As such, how should the manual analysis proceed? Should it rotate the screen whenever an event is executed? And if no failure is encountered nor detected should it be considered a true negative? This would lead to an explosion of the number of true negatives and would deem the manual validation infeasible.

Therefore, only the true positives, false positives and false negatives were analysed: the failures detected by iMPAcT and the failures detected by the manual exploration were recorded and then matched: if both detected a failure it is a *true positive*, if iMPAcT detected a failure but the manual exploration did not it is *false positive* and if the manual exploration detected a failure but iMPAcT did not then it is a *false negative*.

Consequently, only the sensitivity and $PV+$ metrics were considered due to the problems that arise when identifying the true negatives. Moreover, iMPAcT's main objective is to identify failures, not necessarily providing confidence that no failures are present when they are not identified.

The experiment consisted in:

1. select a set of applications in which to run iMPAcT;
2. run iMPAcT with each of the patterns;
3. collect the information on the failures found;
4. manually inspect the applications in order to detect the different failures;
5. compare the data obtained by each method in order to identify the true and false positives and the false negatives for each application;
6. calculate the sensitivity and $PV+$ rates.

In order to select the applications an initial selection of five different categories from the Play Store was performed: Books & References, Business, Comics, Finance and Health & Fitness. These categories were chosen in order to ensure a wide range of different characteristics amongst the applications.

Each of these categories was then analysed in order to select five applications, summing up to a total of twenty five applications. This selection was fulfilled in May 2016 and was based on the top applications presented by Google's Play Store at the time for each of the aforementioned categories. For an application to be selected it would have to fulfil all of the following requirements:

- it is free;
- it is in English;
- it does not depend on other applications;
- it does not require a login to interact with the application;
- it does not require access to the device's camera;
- it has been downloaded at least five hundred thousand times;
- it runs on a Nexus 5 with Android 6.0;
- it is a native application;
- it can be read by *uiautomatorviewer*;
- it does not require access to the device's contacts or documents;
- it presents *at least* one of the following characteristics:
 - it responds to the rotation of the device;
 - it presents a side drawer;
 - it presents at least a set of tabs.

Besides these requirements, a minimum of one hundred thousands votes was an initial requirement. However, in some categories this requirement had to be disregarded. Thus, this was considered a plus but not necessary.

The final set of selected applications is presented on Table 5.9.

After selecting the applications, iMPAcT was run on each of them three times (in order to reduce the effect of the randomness) and each execution was limited to twenty minutes (which is the highest amount of time taken by an execution in Section 5.3 and was only surpassed by one of the applications in Section 5.4). This limit was imposed to ensure the experiment took a feasible amount of time while still being able to explore a considerable part of the application in each run.

Table 5.9: Final applications selection

App	Version
Books & References	
JW Library	1.7.1
Scribd - A world of Books	4.5
Bible	6.9.0
Google Play Books	3.9.37
Wikipedia	2.2.147-r-2016-06-06
Business	
Job Search	2.9
File Commander - File Manager	3.8.14500
Find Work Offers - Trovit Jobs	4.2.4
BZ Reminder	1.5.1
Call Blocker Free - BlackList	5.2.22.00
Comics	
Marvel Comics	3.8.3.38302
Draw Cartoon 2	0.3.35
Draw Anime - Manga Tutorials	1.3.2
DC Comics	3.8.3.38303
Comics	3.9.1.39108
Finance	
Money Lover - Money Manager	android-3.3.9
Investing.com Shares & Forex	2.7.28
Moneyfy - Money Manager	1.7.3
MSN Money - Stock Quotes	1.1.0
Meta Trader 4	400.954
Health & Fitness	
Clue - Period Tracker	2.2.8
Pedometer	5.1.5
7 Minute Workout	1.29.64
Abs Workout	8.11.1
S Health	4.8.1.0013

Considering it was important to maximise the number of different screens analysed when running iMPAcT during this experiment, the selected exploration algorithm was unlike the previously presented experiments, *priority to not executed*.

In order to avoid interferences from the other patterns, iMPAcT tested each of them separately, *i.e.* it was run three times for each pattern in each application totalling nine runs per application. This Section does not report the results for the resource dependency pattern as no failure was found.

During these executions, the information of when a Test Pattern was applied and when a failure was detected was recorded.

Finally, a manual inspection of each application was performed. In this inspection, the pre-conditions of the test patterns were not considered, *i.e.* whenever a change occurred in the screen, the side drawer was opened and analysed, the device was rotated and the tabs present were analysed according to their quantity, position and whether it was possible to move among them by horizontally swiping the screen. The idea behind this was trying to exercise the application as thoroughly as possible to avoid missing any failures.

The results obtained by these two processes were then compared in order to obtain a table similar to Table 5.8 for each application and for each pattern. The results obtained for each application and for each pattern can be found in Attachment A.

Tables 5.10, 5.11 and 5.12 present the final results obtained for side drawer, orientation and tab patterns, respectively, *i.e.* the total number of true positives, false positives and false negatives obtained for all the applications.

Table 5.10: Final results of true/false positives/negatives for the Side Drawer Pattern

		iMPAcT	
		detects a failure	does not detect a failure
Manual inspection			
	is failure	5	8
	is not failure	0	-

Table 5.11: Final results of true/false positives/negatives for the Orientation Pattern

		iMPAcT	
		detects a failure	does not detect a failure
Manual inspection			
	is failure	59	21
	is not failure	5	-

Table 5.12: Final results of true/false positives/negatives for the Tab Pattern

		iMPAcT	
		detects a failure	does not detect a failure
Manual inspection			
	is failure	3	5
	is not failure	0	-

These results enable the analysis on the degree of confidence the iMPAcT tool provides when identifying a failure: is it capable of finding failures and does it correctly

characterise them. The sensitivity and PV+ values for each pattern are presented in Table 5.13.

Table 5.13: Final Results

Metric	Side Drawer	Orientation	Tabs
Sensitivity	0.385	0.7375	0.375
<i>PV+</i>	1	0.922	1

The major threat to validity of this experiment is the fact that the validation of the results was processed manually, being possibly subject to errors in the analysis. However, if another automatic process was used, the results would also depend on the reliability of that process.

5.6 Visualisation of Results

During the execution, iMPAcT builds a model of the application in the form of a FSM as explained in Section 3.7. Each state represents an activity of the AUT and each arrow represents an event. The states can either be represented as ovals or by the screenshot of the screen they represent if the corresponding option is selected in the main window of the iMPAcT tool (Figure 4.2 in Section 4.5).

Figure 5.5 depicts the state machine obtained while exploring the Tippy Tipper application. The states are screenshots taken whenever a new screen was detected during the exploration. Screen 0 is the initial state of the application.

As stated in Section 3.7, this FSM is extremely useful for comprehending the behaviour of the application, depicting its different states and how one can navigate within the application and may be useful to either reproduce the failures detected or to retrace the steps taken after the failure is allegedly corrected. For example, the path that leads to a failure (in the Orientation Pattern) is highlighted in red bold in Figure 5.5. In this case, in Screen 0 (the initial state) the user clicks on the "Calculate!" button (event 1) going to Screen 1. If the device is rotated at this state the "Round Down" and "Round Up" buttons disappear.

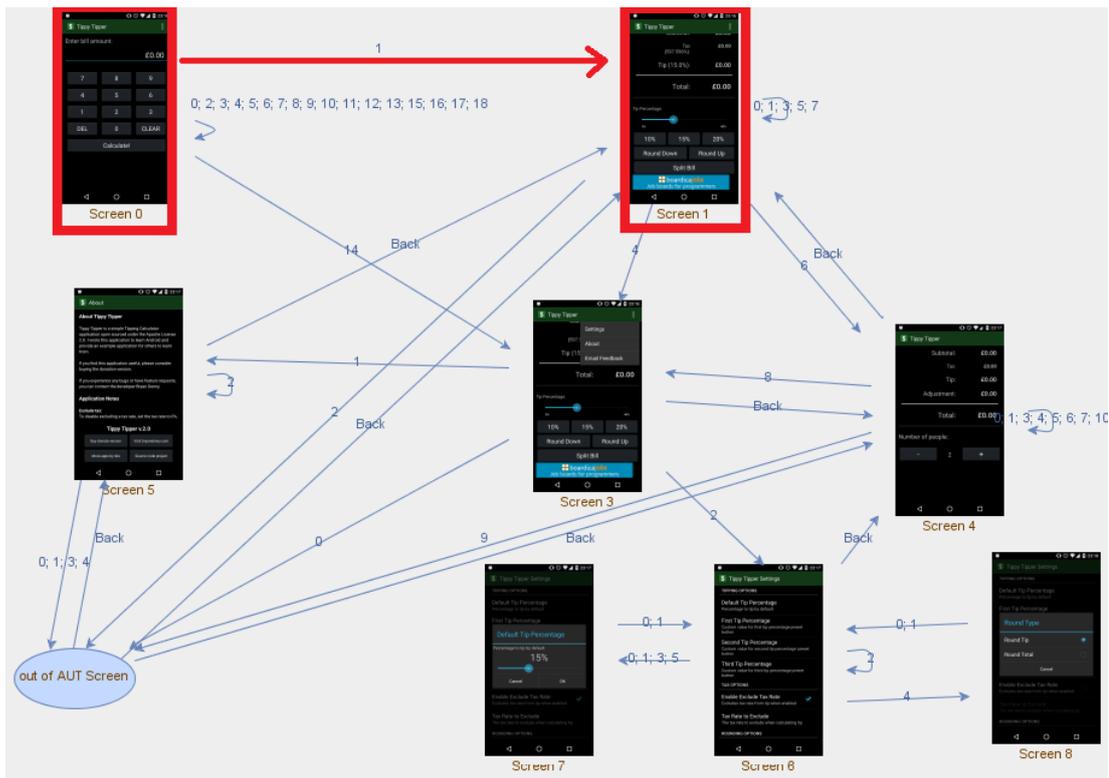


Figure 5.5: State Machine of the Tippy Tipper application

5.7 Conclusions

This Chapter presented three different experiments that were performed in order to answer the proposed research questions. These research questions, more specifically 1, 2, 5 and 6, were defined with the research hypothesis (defined in Section 1.3) in mind. They enable the analysis of the capacity of iMPAcT for finding failures and if its results are reliable. Furthermore, RQ3 and RQ4 were defined to analyse the effect of the execution configuration on the results. The discussion on the results obtained is on Chapter 6.

Chapter 6

Discussion

Chapter 5 defined the research questions important for validating the research hypothesis and presented the three experiments defined to answer them. It also presented the results obtained with each of the experiments. On the other hand, this Chapter discusses the results obtained and answers the research questions.

6.1 RQ1: Is iMPAcT able to identify failures in Android Mobile Applications?

In order to answer this research question the experiment described in Section 5.3 was defined. This experiment applied iMPAcT to five different applications in order to verify if any failures were detected.

The results obtained with the experiments performed show that the iMPAcT tool is able to effectively explore an Android application while testing its UI Patterns. This experiment also showed that the iMPAcT tool is capable of successfully identifying and testing UI Patterns present in mobile applications. Moreover, some of the detected failures were enumerated and illustrated. It is important to state, though, that some failures detected by iMPAcT are in fact false positives. For instance, Figure 5.3 depicts a detected failure whose cause is the disappearance of the option *Search* from the menu when the screen is rotated. However, after a more thorough analysis of the screen in landscape, it is possible to verify that the *Search* option is present on the screen in the Action Bar. Nevertheless, when using UiAutomation to read the screen only the menu elements are identified (the root element of the screen is actually the root element of the menu) and, thus, it is not possible to implement an heuristic to minimise this sort of false positives. However, the remaining failures reported correspond to actual failures in the application.

None of the applications tested presented failures in the *Resource Dependency* pattern when configured for *Wifi*.

These results enable us to respond affirmatively to **RQ1**: iMPAcT is capable of identifying failures in Android mobile applications.

6.2 RQ2: Is iMPAcT's performance affected by the size of the applications?

This research question can be divided in two parts:

- a) is the coverage acceptable, *i.e.* is it capable of exploration the majority of the identified events?
- b) does the whole process take a reasonably amount of time?

The results obtained by the experiment described in Section 5.3 can also be used to answer this RQ.

In order to analyse the coverage, the percentage of executed events was recorded and analysed: the higher the percentage, the better the coverage. This percentage does not regard the actual amount of events that can be executed in the application but the number of different events that are identified during the exploration. Hence, two types of percentages were recorded: the percentage of executed events (*% of events* on Table 5.3), which considers the amount of events executed in a single execution in regards to the amount identified in that same run, and the percentage of executed events in regards to the total of events identified (*% of events over all explorations* on Table 5.3), which considers the amount of events executed in a single execution in regards to the maximum amount identified in all five executions. The difference between these percentages is one of the main reasons we decided to run iMPAcT five times in each of the applications.

The applications tested in this experiment have different sizes: even though Tomdroid only occupies 1MB of space, Google Slides occupies more than 83MB. Hence, it is also possible to analyse how the size of the applications may affect the results.

When reducing the analysis to a single execution (*% of events*), every application presents values over 65%. In fact, for three of them this value is over 75% and for the Tippy Tipper application it is over 90%. These values point to a good coverage from iMPAcT.

Even though it is not possible to state that the larger the application the lower the percentage as, for instance, the Tomdroid application is smaller than the Tippy Tipper

application and still presents a lower percentage, it is possible to verify that smaller applications tend to present higher percentages: the smallest applications present the highest coverage percentages and the largest applications present the lowest coverage percentages.

When broadening the analysis to include the total number of events identified over all five executions, the percentages (% of events over all executions) present lower values (the minimum of which is 45%). Nevertheless, the majority present over 50% of coverage, with Tippy Tipper presenting, again, the higher value: 74%. Even though the results are not as good as the previous ones, which is to be expected, they still indicate good coverage from iMPAcT.

When analysing the percentages against the size of the applications, the conclusion is the same as before: even though there is not a direct relation, there is a tendency to obtain better results with smaller applications than with larger ones.

With this data it is possible respond to the first part of **RQ2**: the coverage of iMPAcT is affected by the size of the application but even with big applications, it is still capable of obtaining a good coverage (over 60% and over 50%, depending on the denominator being considered). Nevertheless, the exploration algorithm can still be improved.

Apart from the percentages of executed events, the amount of time the execution took was also recorded in order to respond to the second part of RQ2.

First of all, it is important to notice that the amount of time required to explore the applications never exceeded twenty minutes. Considering the approach does not require any manual effort enabling the testers to refocus their attention after starting the execution, the time iMPAcT takes is feasible even for larger applications. Still, the user can stop the execution at any time by pressing the device's home button.

The relation between time of execution and size of application is similar to the previously analysed relationships: smaller applications tend to take less time than larger applications even if in some cases this does not happen (*e.g.*, Google Calendar is larger than Book Catalogue and still takes less time to be explored).

In conclusion, the overall performance of iMPAcT in terms of both coverage and execution time is good but it can be affected by the size of the application.

6.3 RQ3: Does the exploration algorithm used affect the results, performance and coverage of iMPAcT?

In order to answer this research question, the experiment presented in Section 5.4 was defined. In this experiment, iMPAcT was run on two different applications with the three

exploration algorithms recording the number of failures detected, the time of execution (Tables 5.4 and 5.6) and the percentage of events executed in regards to the number of events identified in a single execution (% of events) and in regards to the number of events identified in the executions with the same exploration algorithm (% of events over exploration algorithm) (Tables 5.5 and 5.7). As explained later (in Section 6.4) this experiment was also run varying the order in which the patterns are identified and tested. As such a third coverage metric was calculated: the percentage of executed events in regards to the events identified over all six configurations (three exploration algorithm times two testing orders).

By analysing the results obtained it is possible to state that:

- the exploration algorithms 2 and 3 take more time than exploration algorithm 1;
- even though in the TomDroid application exploration algorithm 2 and 3 take a similar amount of time, in the Book Catalogue application, the third algorithm takes significantly longer to finish;
- the number of failures detected is not affected by the algorithm chosen in Tomdroid but it is highly affected in Book Catalogue. This happens because, unlike Book Catalogue, Tomdroid presents only a few failures, which are easily detected by either of the algorithms;
- the % of events increases with the complexity of the exploration algorithm ($1 < 2 < 3$);
- the % of events over exploration algorithm is also affected by the exploration algorithm, even though this difference is less significant between algorithms 2 and 3;
- the % of events over all explorations is significantly affected by the exploration algorithm;

The difference in the execution time between exploration algorithm 1 and exploration algorithms 2 and 3 is to be expected as the first one executes a much lower number of events (identifies less events and executes a smaller percentage of events). The difference in the overall results between algorithm 1 and algorithms 2 and 3 shows that the two last algorithms enable a more thorough exploration of the application, which leads to more failures being found.

Moreover, exploration algorithm 3 provides a more thorough exploration as it identifies more events and still executes a higher percentage of events. It also detects more failures. However, it takes much longer to execute than the other exploration algorithms.

The results obtained in this experiment make it possible to conclude that the exploration algorithm largely affects the results (the more complex the algorithm the higher number of failures detected), the performance (the more complex the algorithm, the longer the execution takes) and the coverage (the more complex the algorithm, the higher the percentage of events executed), *i.e.* the answer to RQ3 is affirmative.

6.4 RQ4: Does the order in which the patterns are identified and tested affect the results, performance and coverage of iMPAcT?

The experiment presented in Section 5.4 can also be used to answer this question. Apart from the data identified in Section 6.3, this experiment also explored each of the applications with variations on the order in which the patterns were identified and tested. Only the side drawer and the orientation patterns were considered. Thus, iMPAcT was run for side drawer-orientation and orientation-side drawer. In order to make the experiment as complete as possible iMPAcT was run with these two orders for each of the exploration algorithms (six configurations). The metrics that matter for this research question are the percentages of executed events in regards to the events identified on a single execution (% of events), the percentage of executed events in regards to the maximum number of events identified for the same configuration (% of events over configuration) and the percentage of executed events in regards to the maximum number of events identified in all executions (% of events over all executions).

Analysing the results it is possible to state that:

- the number of failures found in the orientation pattern is higher when testing the orientation first;
- the execution time is not affected by the order in which the patterns are tested;
- the % of events is not significantly affected by the testing order;
- there is no correlation between the testing order and the % of events over configuration: sometimes the coverage is higher when first testing side drawer sometimes it is higher when first testing orientation;
- In general, the % of events over all executions is not affected by the testing order.

Furthermore, a more thorough analysis of the report makes it possible to conclude that, when testing the orientation pattern before testing the side drawer pattern, the failure “when rotating the screen after clicking *Add Book* menu, the pop-up disappears” is detected (as depicted in Figure 5.1), while the opposite does not occur.

In conclusion, it is possible to state that the testing order affects the results obtained by iMPAcT as, in general, it was possible to detect more failures when testing the orientation pattern before the side drawer pattern than the other way around. However, neither the performance nor the coverage were significantly affected. As such, the answer to RQ2 (Does the order in which the patterns are identified and tested affect the results, performance and coverage of iMPAcT?) is divided in two: the results are affected but neither the performance nor the coverage are.

6.5 RQ5: Are the results provided by iMPAcT sensitive, *i.e.* are the existing failures properly identified by iMPAcT?

According to the results presented in Section 5.5, the answer to this research question depends on the pattern being analysed. When testing the orientation pattern iMPAcT detects almost 75% of existing failures. However, when testing the side drawer and tab patterns, this percentage drops to 40%.

After a more careful analysis of the results presented in Attachment A, it is possible to conclude that:

1. most of the failures that were not detected in the side drawer pattern are due to not correctly identifying the presence of a side drawer;
2. all the failures that were not detected in the tab pattern are due to not correctly identifying the presence of a set of tabs;
3. some of the failures that were not identified by the orientation pattern are due to not identifying a screen as a new activity and, thus, not applying the rotation pattern;
4. some of the failures that were not identified by the orientation pattern are caused by the time limit imposed for the experiments, *i.e.* that part of the application is never explored.

The main reason for these misidentifications is the necessity of implementing heuristics to identify the UI Patterns as often developers do not follow any implementation standard even though Google presents guidelines for some of them [Android, 2016a,c].

Assuming that the iMPAcT tool would identify a failure if the test was applied (*i.e.* if the UI Pattern was identified), correcting the aforementioned identifications would result in the following sensitivity values:

- Side Drawer: 0.8462
- Orientation: 0.7875
- Tab: 1

Apart from the Orientation pattern the results would improve radically. Hence, it is important to improve the heuristics responsible for the identification of the UI Patterns.

In summary, the answer to RQ5 is affirmative for the orientation pattern but not yet for the side drawer and tab patterns. However, further refining the heuristics of the corresponding UI Patterns would significantly improve the results.

6.6 RQ6: Does iMPAcT reliably identify failures, *i.e.* are the failures detected by iMPAcT actually failures?

The results presented in Section 5.5 are very good, with all the failures detected in the side drawer and tab patterns being actual failures and with the confidence for the orientation pattern being higher than 90%.

Hence, the answer to this research question is affirmative, the tester can rely on the results obtained by iMPAcT when it identifies a failure.

6.7 Conclusions

Considering the experiments performed, the results obtained show that the iMPAcT tool is able to effectively explore an Android application while testing its UI Patterns with a high coverage of the number of identified events. In fact, the minimum obtained is 45% (considering the maximum number of events identified in all executions) going up to 93% (considering the coverage on a single execution). However, the results obtained by iMPAcT can be affected by the exploration algorithm used and by the order in which the patterns are detected. The best algorithm to be used depends on the goal intended by the user. If the objective is to explore each activity as thoroughly as possible the *priority to not executed and list items* algorithm should be chosen. On the other hand if the main purpose is to explore a wider range of activities than the *priority to not executed* algorithm is the better choice.

iMPAcT is able to correctly identify failures in all implemented patterns except for the Resource Dependency one. However, even when manually testing the applications for the experiment presented in Section 5.5, no failure was detected.

The quality of the results of the tests depends on the pattern being tested, being the orientation pattern the one with the best results overall. When testing the side drawer and the tabs, iMPAcT's capacity of detecting failures is damaged by the lack of standards used in the implementation of the corresponding UI Patterns, resulting in some failures not being detected because the test is not applied. On the other hand, when iMPAcT detects a failure in one of these two patterns, the confidence of it actually being a failure is 100%. The capacity of finding failures on the orientation pattern is much higher than on the other patterns, with a confidence of nearly 75% and, even though the confidence of a detected failure actually being a failure is lower than on the other patterns, it is still over 90%.

The duration of the execution depends on the algorithm being used and on the application and it may be too lengthy: in the experiment presented in Section 5.3 one of the applications took twenty minutes to be explored, in the one in Section 5.4 one of the applications took thirty minutes and in the one described in Section 5.5 some applications were not fully explored within the imposed limit of twenty minutes. However, this is not an issue as the process is fully automatic and does not require any supervision.

Chapter 7

Conclusions

The iMPAcT tool combines MBT with automatic crawling. One of the main challenges of MBT is the effort needed to manually build the model, even when disregarding the error proneness of the process. An example of this is the work of Costa *et al.* [Costa et al., 2014] in which they automatically generate test cases for Android applications based on a model they manually built using their own Domain-Specific Language (described in [Moreira & Paiva, 2014a]). Some approaches use reverse engineering techniques in order to obtain a model of the application that is built according to what they want to test. For instance, through reverse engineering techniques Franke *et al.* [Franke et al., 2011] obtain a model of the lifecycle of an iOS application and Yang *et al.* [Yang et al., 2013] obtain a model representing the event-based behaviour of an Android application.

The approach presented in this document also uses reverse engineering to tackle this issue. Unlike Franke *et al.* (and alike Yang *et al.*) we obtain a model that represents the event-based behaviour of the application. On the other hand, the process is fully dynamic unlike Yang *et al.*'s, which follows a hybrid reverse engineering approach. Moreover, our approach also tackles the second major issue of MBT: the explosion of test cases. This explosion depends on the specificity of the states of the model, *i.e.* if the states are too specific there is an explosion of the number of test cases but if they are not specific enough then the test cases do not accurately test the application. iMPAcT solves this problem by focusing on certain parts of the application, the UI Patterns, and defining test strategies to test them whenever they are detected. A few other approaches have also explored the presence of patterns on a mobile application. However, only Costa *et al.* apply them in a similar way to iMPAcT.

When automatically exploring applications (also known as crawling), most approaches apply random testing (also known as monkey testing) or a variety of it, which ultimately makes them capable of only finding crashes due to the lack of a test oracle. An example

of a crawler that has been successful in the community is the Android Ripper developed by Amalfitano *et al.*[Amalfitano et al., 2014]. Apart from only being capable of finding crashes, this tool presents a few other differences from iMPAcT:

- the Ripper obtains several models from the crawling process, such as FSMs, event-flow graphs and UML sequence diagrams, while the iMPAcT tool only obtains a FSM of the application's behaviour;
- even though the Ripper does not require access to the source code of the application for the crawling process, they need to instrument the application in order to obtain the different models previously mentioned. The iMPAcT tool, on the other hand, does not need to instrument nor access the source code of the application in any of its phases;
- the Ripper generates JUnit tests while the iMPAcT tool saves the logs of the execution, *i.e.* the execution logs, that may be transformed into tests in the future;
- The Ripper only detects crashes while the iMPAcT tool can also detect other types of failures;
- The Ripper uses the GUI to analyse and test the AUT but it does not focus on testing the GUI of the application.

Even though the ultimate goal of both tools is to automatically test Android applications, the artefacts produced are different, with Ripper producing a higher variety of models and with iMPAcT being able to identify a higher variety of failures. Moreover, iMPAcT focuses on testing the GUI of the application while the Ripper focuses on testing the application through its GUI alike, in fact, the vast majority of GUI testing approaches.

In summary, the contributions of this work are:

1. a catalogue of patterns containing UI Patterns and the corresponding test strategies (Test Patterns) that can be used to test any mobile application;
2. an approach to systematically and iteratively explore the GUI of a mobile application and to test the recurring behaviour it presents (the UI Patterns defined on the catalogue);
3. a tool, iMPAcT, that implements the approach applying it to Android applications, obtaining the detected GUI failures, the execution log and a FSM representing the event-based behaviour of the GUI of the application.

The experiments described in Chapter 5 and discussed in Chapter 6 are useful for restating the presence of patterns on mobile applications and for proving that it is possible to take advantage of such patterns to test a mobile application while exploring it in a reverse engineering process. As such it is possible to state that the research hypothesis presented in Section 1.3 (*Mobile applications have generic recurring behaviour, independent of their specific domain, that may be tested automatically by combining reverse engineering with testing within an iterative process*) is valid.

The more pressing aspect of the future work is the improvement of the catalogue of patterns as the more patterns are defined, the higher the variety of issues that can be detected and, thus, corrected. These patterns could even include aspects related to security or performance, increasing the range of tests performed. Moreover, the exploration can still be improved by defining different algorithms. Improvements on the presentation of the results could also be studied.

Apart from these improvements, a few extra features would be useful to improve the quality of iMPAcT. An aspect that could help the tester to verify if a failure has actually disappeared after being corrected is enabling iMPAcT to read an execution log (or part of it) and to re-execute them on the application in order to re-test the previously bugged UI Pattern. Moreover, the usefulness of the model obtained would increase if it was used to model check the application, alike what was done in [Coimbra Morgado et al., 2012].

References

- Adamsen, C. Q., Mezzetti, G., & Møller, A. (2015). Systematic Execution of Android Test Suites in Adverse Conditions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*, (pp. 83–93). Baltimore, MD, USA: ACM.
- Alexander, C. W., Ishikawa, S., Silverstein, M., & Jacobson, M. (1977). *A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure)*. Oxford University Press, 1 ed.
- Amalfitano, D., Amatucci, N., Fasolino, A. R., Gentile, U., Mele, G., Nardone, R., Vitorini, V., & Marrone, S. (2014). Improving Code Coverage in Android Apps Testing by Exploiting Patterns and Automatic Test Case Generation. In *Proceedings of the 2014 International Workshop on Long-term Industrial Collaboration on Software Engineering (WISE '14)*, (pp. 29–34). Vasteras, Sweden: ACM.
- Amalfitano, D., Fasolino, A., & Tramontana, P. (2011). A GUI Crawling-Based Technique for Android Mobile Application Testing. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, (pp. 252–261). Berlin, Germany: IEEE.
- Amalfitano, D., Fasolino, A. R., Tramontana, P., De Carmine, S., & Imperato, G. (2012a). A toolset for GUI testing of Android applications. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, (pp. 650–653). Trento, Italy: IEEE.
- Amalfitano, D., Fasolino, A. R., Tramontana, P., De Carmine, S., & Memon, A. M. (2012b). Using GUI ripping for automated testing of Android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*, (pp. 258–261). Essen, Germany: ACM Press.
- Amalfitano, D., Fasolino, A. R., Tramontana, P., Ta, B., & Memon, A. (2015). Mobi-GUITAR: Automated Model-Based Testing of Mobile Apps. *IEEE Software*, 32(5), 53–59.
- Amer, M., Goldstein, M., & Abdennadher, S. (2013). Enhancing One-class Support Vector Machines for Unsupervised Anomaly Detection. In *Proceedings of the ACM SIGKDD Workshop on Outlier Detection and Description (ODD '13), ODD '13*, (pp. 8–15). Chicago, Illinois: ACM.
- Anand, S., Naik, M., Harrold, M. J., & Yang, H. (2012). Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium*

- on the Foundations of Software Engineering (FSE '12)*, (pp. 59:1—59:11). Cary, North Carolina: ACM.
- Andoni, A. (2005). LSH Algorithm and Implementation (E2LSH). URL <http://www.mit.edu/~andoni/LSH/> Accessed on 2016-03-14
- Android, G. (2008). Announcing the Android 1.0 SDK, release 1. URL <http://goo.gl/5PSQHj> Accessed on 2016-02-16
- Android, G. (2015a). Android - What To Test. URL <http://goo.gl/AL22tJ> Accessed on 2015-02-14
- Android, G. (2015b). Android Navigation Drawer. URL <http://goo.gl/nnJOoj> Accessed on 2015-07-01
- Android, G. (2015c). Up and running with material design. URL <https://goo.gl/GmsJSJ> Accessed on 2015-08-01
- Android, G. (2016a). Creating a Navigation Drawer. URL <https://developer.android.com/training/implementing-navigation/nav-drawer.html> Accessed on 2016-06-27
- Android, G. (2016b). Pure Android. URL <http://goo.gl/LqNPyS> Accessed on 2016-01-08
- Android, G. (2016c). TabLayout. URL <https://developer.android.com/reference/android/support/design/widget/TabLayout.html> Accessed on 2016-07-27
- Android, G. (2016d). Tabs. URL <https://www.google.com/design/spec/components/tabs.html> Accessed on 2016-01-08
- Appium (2016). Appium. URL <http://appium.io/> Accessed on 2016-02-01
- Apple (2007). Apple Reinvents the Phone with iPhone. URL <https://goo.gl/AoFdyx> Accessed on 2015-02-16
- Apple (2016). Common App Rejections. URL <https://developer.apple.com/app-store/review/rejections/> Accessed on 2016-07-01
- Avancini, A., & Ceccato, M. (2013). Security testing of the communication among Android applications. In *Automation of Software Test (AST), 2013 8th International Workshop on*, (pp. 57–63). San Francisco, California: IEEE Press.
- Azim, T., & Neamtiu, I. (2013). Targeted and depth-first exploration for systematic testing of android apps. *ACM SIGPLAN Notices*, 48(10), 641–660.
- Batyuk, L., Herpich, M., Camtepe, S. A., Raddatz, K., Schmidt, A.-D., & Albayrak, S. (2011). Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within Android applications. In *Malicious and Unwanted Software (MALWARE), 2011 6th International Conference on*, (pp. 66–72). Fajardo, Puerto Rico: IEEE.

- Bell, T. (1999). The concept of dynamic analysis. *ACM SIGSOFT Software Engineering Notes*, 24(6), 216–234.
- Binkley, D. (2007). Source Code Analysis: A Road Map. In *Future of Software Engineering, 2007. FOSE '07*, (pp. 104–119). Minneapolis, MN: IEEE.
- Blei, D. M., Ng, A. Y., & Jordan, M. I. (2003). Latent dirichlet allocation. *Journal of Machine Learning Research*, 3(Jan), 993–1022.
- Briand, L., Labiche, Y., & Leduc, J. (2006). Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software. *IEEE Transactions on Software Engineering*, 32(9), 642–663.
- Butcher, N., & Developers, A. (2016). Android Design in Action: Navigation Anti-Patterns. URL <https://www.youtube.com/watch?v=Sww4omntVjs> Accessed on 2016-01-08
- Butcher, N., & Nurik, R. (2016). Android Design in Action: Navigation Anti-Patterns. URL <http://www.allreadable.com/ff647Z8N> Accessed on 2016-01-08
- Canfora, G., & Di Penta, M. (2007). New Frontiers of Reverse Engineering. In *Future of Software Engineering, 2007. FOSE '07*, (pp. 326 – 341). Minneapolis, MN, USA.
- Capgemini, Hp, & Sogeti (2014). World Quality Report 2014-15. Tech. rep., Capgemini.
- Capgemini, HP, & Sogeti (2015). World Quality Report 2015-16. Tech. rep., Capgemini.
- Carbonell, J. G., Michalski, R. S., & Mitchell, T. M. (1983). Machine Learning: An Artificial Intelligence Approach. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.) *Machine Learning: An Artificial Intelligence Approach*, chap. An Overview, (pp. 3–23). Berlin, Heidelberg: Springer Berlin Heidelberg.
- Chen, K., Liu, P., & Zhang, Y. (2014). Achieving accuracy and scalability simultaneously in detecting application clones on Android markets. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*, (pp. 175–186). Hyderabad, India: ACM.
- Chen, T. Y., Leung, H., & Mak, I. K. (2005). Adaptive Random Testing. *Advances in Computer Science - ASIAN 2004. Higher-Level Decision Making*, 3321, 320–329.
- Chikofsky, E., & Cross, J. (1990). Reverse Engineering and Design Recovery: a Taxonomy. *IEEE Software*, 7(1), 13–17.
- Choi, W., Necula, G., & Sen, K. (2013). Guided GUI testing of android apps with minimal restart and approximate learning. *ACM SIGPLAN Notices*, 48(10), 623–640.
- Coimbra Morgado, I., & Paiva, A. C. R. (2015a). Testing approach for mobile applications through reverse engineering of UI patterns. In *Sixth International Workshop on Testing Techniques for Event Based Software*.

- Coimbra Morgado, I., & Paiva, A. C. R. (2015b). The iMPAcT Tool: Testing UI Patterns on Mobile Applications. In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015)*. Lincoln, NE, USA.
- Coimbra Morgado, I., & Paiva, A. C. R. (2016). Impact of execution modes on finding Android failures. *The 7th International Conference on Ambient Systems, Networks and Technologies*, 83, 284–291.
- Coimbra Morgado, I., Paiva, A. C. R., & Pascoal Faria, J. (2012). Dynamic Reverse Engineering of Graphical User Interfaces. *International Journal On Advances in Software*, 5(3 and 4), 224–236.
- Corbi, T. A. (1989). Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2), 294–306.
- Costa, P., Paiva, A. C. R., & Nabuco, M. (2014). Pattern Based GUI Testing for Mobile Applications. In *9th International Conference on the Quality of Information and Communications Technology (QUATIC 2014)*, (pp. 66–74). Guimarães, Portugal: IEEE.
- Dar, M. A., & Parvez, J. (2014). Enhancing security of Android & IOS by implementing need-based security (NBS). In *Control, Instrumentation, Communication and Computational Technologies (ICCICCT), 2014 International Conference on*, (pp. 728–733). Kanyakumari, India: IEEE.
- Deng, L., Mirzaei, N., Ammann, P., & Offutt, J. (2015). Towards mutation analysis of Android apps. In *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*, (pp. 1–10). Graz, Austria: IEEE.
- Developer, A. (2016). iOS Human Interface Guidelines. URL <https://developer.apple.com/ios/human-interface-guidelines/ui-bars/navigation-bars/> Accessed on 2016-01-16
- Developers, A. (2012). Android SDK Tools, Revision 21. URL <http://android-developers.blogspot.pt/2012/11/android-sdk-tools-revision-21.html> Accessed on 2016-02-01
- Developers, A. (2014). Espresso 2.0. URL <https://plus.google.com/+AndroidDevelopers/posts/jHXFkebKjEb> Accessed on 2016-02-01
- Developers, A. (2015a). Testing UI for Multiple Apps. URL <http://developer.android.com/training/testing/ui-testing/uiautomator-testing.html> Accessed on 2016-03-14
- Developers, A. (2015b). UIAutomator 2.0. URL <https://plus.google.com/+AndroidDevelopers/posts/WCWANrPkRxg> Accessed on 2015-03-15
- Easterbrook, S., Singer, J., Storey, M.-A., & Damian, D. (2008). *Guide to Advanced Empirical Software Engineering*. London: Springer-Verlag New York, Inc.
- Eilam, E. (2005). *Reversing: Secrets of Reverse Engineering*. Indianapolis, Indiana: Wiley, 1st editio ed.

- Erlikh, L. (2000). Leveraging legacy system dollars for e-business. *IT Professional*, 2(3), 17–23.
- Foundation, e. S. (2016). Selendroid - Selenium for Android. URL <http://selendroid.io/> Accessed on 2016-07-07
- Fowler, M. (1997). *Analysis Patterns - Reusable Object Models*. Addison-Wesley.
- Franke, D., Elsemann, C., Kowalewski, S., & Weise, C. (2011). Reverse Engineering of Mobile Application Lifecycles. In *2011 18th Working Conference on Reverse Engineering*, (pp. 283–292). Limerick, Republic of Ireland: IEEE.
- Franke, D., Kowalewski, S., Weise, C., & Prakobkosol, N. (2012a). Testing Conformance of Life Cycle Dependent Properties of Mobile Applications. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, (pp. 241–250). Montreal, QC, Canada: IEEE.
- Franke, D., Royé, T., & Kowalewski, S. (2012b). AndroLIFT: A Tool for Android Application Life Cycles. In *VALID 2012: The Fourth International Conference on Advances in System Testing and Validation Lifecycle*, (pp. 28–33).
- Fratantonio, Y., Machiry, A., Bianchi, A., Kruegel, C., & Vigna, G. (2015). CLAPP: characterizing loops in Android applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*, (pp. 687–697). Bergamo, Italy: ACM Press.
- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley professional computing series, 1st ed.
- Gao, J., Bai, X., Tsai, W.-T., & Uehara, T. (2014). Mobile application testing - a Tutorial. *Computer*, 47(2), 46–55.
- Gartner (2013). Gartner Says Annual Smartphone Sales Surpassed Sales of Feature Phones for the First Time in 2013. URL <http://www.gartner.com/newsroom/id/2665715> Accessed on 2016-03-21
- Gartner (2015). Gartner Says Smartphone Sales Surpassed One Billion Units in 2014. URL <http://www.gartner.com/newsroom/id/2996817> Accessed on 2016-04-18
- Gartner (2016a). Gartner Says Emerging Markets Drove Worldwide Smartphone Sales to 15.5 Percent Growth in Third Quarter of 2015. URL <http://www.gartner.com/newsroom/id/3169417> Accessed on 2016-04-18
- Gartner (2016b). Gartner Says Worldwide Smartphone Sales Grew 9.7 Percent in Fourth Quarter of 2015. URL <http://www.gartner.com/newsroom/id/3215217> Accessed on 2016-04-18
- Ghezzi, C., Jazayeri, M., & Mandrioli, D. (2002). *Fundamentals of Software Engineering*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2 ed.

- Gomez, L., Neamtiu, I., Azim, T., & Millstein, T. (2013). RERAN: Timing- and touch-sensitive record and replay for Android. In *2013 35th International Conference on Software Engineering (ICSE)*, (pp. 72–81). San Francisco, CA, USA: IEEE.
- Gorla, A., Tavecchia, I., Gross, F., & Zeller, A. (2014). Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*, (pp. 1025–1035). Hyderabad, India: ACM Press.
- H., V. (2013). Android's Google Play beats App Store with over 1 million apps, now officially largest. URL [{_}id45680](http://www.phonearena.com/news/Androids-Google-Play-beats-App-Store-with-over-1-million-apps-now-officially-largest) Accessed on 2016-07-29
- Holl, K., & Elberzhager, F. (2014). A Mobile-Specific Failure Classification and Its Usage to Focus Quality Assurance. In *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, (pp. 385–388). Verona, Italy: IEEE.
- Hu, C., & Neamtiu, I. (2011). Automating GUI testing for android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test (AST '11)*, (pp. 77–83). Waikiki, Honolulu, HI, USA: ACM.
- Hu, W., Ocateau, D., McDaniel, P. D., & Liu, P. (2014). Duet: Library Integrity Verification for Android Applications. In *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks (WiSec '14)*, (pp. 141–152). Oxford, United Kingdom: ACM Press.
- IEEE (1990). IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, (pp. 1–84).
- Imparato, G. (2015). A combined technique of GUI ripping and input perturbation testing for Android apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, (pp. 760–762). Florence, Italy: IEEE.
- Ingraham, N. (2013). Apple announces 1 million apps in the App Store, more than 1 billion songs played on iTunes radio. URL <http://www.theverge.com/2013/10/22/4866302/apple-announces-1-million-apps-in-the-app-store> Accessed on 2016-07-29
- ISO/IEC (2011). ISO/IEC 25010:2011 - Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. Tech. rep.
- Jensen, C. S., Prasad, M. R., & Møller, A. (2013). Automated testing with targeted event sequence generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013)*, (pp. 67–77). Lugano, Switzerland: ACM.
- Joorabchi, M. E., & Mesbah, A. (2012). Reverse Engineering iOS Mobile Applications. In *2012 19th Working Conference on Reverse Engineering*, (pp. 177–186). Kingston, ON, Canada: IEEE.

- Kagdi, H., Collard, M. L., & Maletic, J. I. (2007). A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(2), 77–131.
- Keng, J. C. J., Jiang, L., Wee, T. K., & Balan, R. K. (2016). Graph-aided directed testing of Android applications for checking runtime privacy behaviours. In *Proceedings of the 11th International Workshop on Automation of Software Test (AST '16)*, (pp. 57–63). Austin, Texas, USA: ACM Press.
- Kuhn, D. R., Kacker, R. N., & Lei, Y. (2010). SP 800-142. Practical Combinatorial Testing. Tech. rep., Gaithersburg, MD, USA.
- Kuhn, H. W. (1955). The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2), 83–97.
- Larman, C. (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 3rd ed.
- Lázaro, M., & Marcos, E. (2005). Research in Software Engineering: Paradigms and Methods. In *17th International Conference on Advanced Information Systems Engineering (CAISE 2005)*, (pp. 517–522). Porto, Portugal.
- Lee, K., Flinn, J., Giuli, T., Noble, B., & Peplin, C. (2013). AMC. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '13)*, (pp. 1–12). Taipei, Taiwan: ACM.
- Lientz, B. P., Swanson, E. B., & Tompkins, G. E. (1978). Characteristics of application software maintenance. *Communications of the ACM*, 21(6), 466–471.
- Liu, Z., Gao, X., & Long, X. (2010). Adaptive random testing of mobile application. In *Computer Engineering and Technology (ICCET), 2010 2nd International Conference on*, vol. 2, (pp. 297–301). Chengdu, China: IEEE.
- Machiry, A., Tahiliani, R., & Naik, M. (2013). Dynodroid: an input generation system for Android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*, (pp. 224–234). Saint Petersburg, Russia: ACM Press.
- Mahmood, R., Esfahani, N., Kacem, T., Mirzaei, N., Malek, S., & Stavrou, A. (2012). A whitebox approach for automated security testing of Android applications on the cloud. In *Automation of Software Test (AST), 2012 7th International Workshop on*, (pp. 22–28). Zurich, Switzerland: IEEE.
- Mariani, L., Pastore, F., & Pezze, M. (2011). Dynamic Analysis for Diagnosing Integration Faults. *IEEE Transactions on Software Engineering*, 37(4), 486–508.
- Mausser, D., Klaus, A., Holl, K., & Zhang, R. (2013). GUI Failures of In-Vehicle Infotainment: Analysis, Classification, Challenges, and Capabilities. *International Journal on Advances in Software*, 6(1&2), 142–154.

- Memon, A. M., Banerjee, I., & Nagarajan, A. (2003). GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. In *Reverse Engineering, 2003. WCRE 2003. Proceedings. 10th Working Conference on*, (pp. 260–269). Victoria, Canada: IEEE.
- Microsoft (2013). Introduction to Instrumentation and Tracing. URL [http://msdn.microsoft.com/en-us/library/aa983649\(VS.71\).aspx](http://msdn.microsoft.com/en-us/library/aa983649(VS.71).aspx) Accessed on 2015-10-02
- Microsoft (2016). Testing apps for Windows Phone 8. URL [https://msdn.microsoft.com/en-us/library/windows/apps/jj247547\(v=vs.105\).aspx](https://msdn.microsoft.com/en-us/library/windows/apps/jj247547(v=vs.105).aspx) Accessed on 2016-07-01
- Moran, K., Linares-Vasquez, M., Bernal-Cardenas, C., Vendome, C., & Poshyvanyk, D. (2016). Automatically Discovering, Reporting and Reproducing Android Application Crashes. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, (pp. 33–44). Chicago, IL, USA: IEEE.
- Moreira, R. M. L. M., & Paiva, A. C. R. (2014a). A GUI Modeling DSL for Pattern-Based GUI Testing PARADIGM. In *9th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE'2014)*. Lisbon, Portugal.
- Moreira, R. M. L. M., & Paiva, A. C. R. (2014b). PBGT tool: an integrated modeling and testing environment for pattern-based GUI testing. In *29th ACM/IEEE international conference on Automated software engineering (ASE 2014)*, (pp. 863–866). New York, New York, USA: ACM Press.
- Muccini, H., di Francesco, A., & Esposito, P. (2012). Software testing of mobile applications: Challenges and future research directions. In *Automation of Software Test (AST), 2012 7th International Workshop on*, (pp. 29–35). Zurich, Switzerland: IEEE.
- Muller, H. A., Jahnke, J. H., Smith, D. B., & Storey, M.-A. (2000). Reverse engineering: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering (ICSE '00)*, (pp. 47–60). Limerick, Ireland: ACM Press.
- Nabuco, M., & Paiva, A. C. R. (2014). Model-Based Test Case Generation for Web Applications. In *14th International Conference on Computational Science and Applications (ICCSA 2014)*.
- Neil, T. (2014). *Mobile Design Pattern Gallery: UI Patterns for Smartphone Apps*. Sebastopol, Canada: O'Reilly Media, Inc., 2nd ed.
- Ng, J. K.-Y., Guéhéneuc, Y.-G., & Antoniol, G. (2010). Identification of behavioural and creational design motifs through dynamic analysis. *Journal of Software Maintenance and Evolution: Research and Practice*, 22(8), 597–627.
- Nguyen, C. D., Marchetto, A., & Tonella, P. (2012). Combining model-based and combinatorial testing for effective test case generation. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA 2012)*, (pp. 100–110). Minneapolis, MN, USA: ACM Press.
- Nilsson, E. G. (2009). Design patterns for user interface for mobile applications. *Advances in Engineering Software*, 40(12), 1318–1328.

- Octeau, D., Jha, S., & McDaniel, P. (2012). Retargeting Android applications to Java bytecode. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*, (pp. 6:1–6:11). Cary, North Carolina: ACM Press.
- OpenQA (2016). SeleniumHQ - Browser Automation. URL <http://www.seleniumhq.org/> Accessed on 2016-07-07
- Ravitch, T., Creswick, E. R., Tomb, A., Foltzer, A., Elliott, T., & Casburn, L. (2014). Multi-App Security Analysis with FUSE. In *Proceedings of the 4th Program Protection and Reverse Engineering Workshop (PPREW-4)*, (pp. 4:1–4:10). New Orleans, LA, USA: ACM.
- Rekoff, M. (1985). On Reverse Engineering. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-15(2), 244 – 252.
- Robotium (2016). Robotium. URL <http://robotium.com/> Accessed on 2016-02-01
- Rothlisberger, D., Greevy, O., & Nierstrasz, O. (2008). Exploiting Runtime Information in the IDE. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, (pp. 63–72). Amsterdam, Netherlands: IEEE.
- Sable, R. G. (2016). Soot - A framework for analyzing and transforming Java and Android Applications. URL <https://sable.github.io/soot/> Accessed on 2016-08-04
- Sacramento, C., & Paiva, A. C. R. (2014). Web Application Model Generation through Reverse Engineering and UI Pattern Inferring. In *9th International Conference on the Quality of Information and Communications Technology (QUATIC 2014)*, (pp. 105–115). Guimarães, Portugal: IEEE.
- Sahami Shirazi, A., Henze, N., Schmidt, A., Goldberg, R., Schmidt, B., & Schmauder, H. (2013). Insights into layout patterns of mobile user interfaces by an automatic analysis of android apps. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '13)*, (pp. 275–284). London, United Kingdom: ACM.
- Sen, K., Marinov, D., Agha, G., Sen, K., Marinov, D., & Agha, G. (2005). CUTE: A concolic unit testing engine for C. *SIGSOFT Software Enging Notes*, 30(5), 263–272.
- Shahriar, H., North, S., & Mawangi, E. (2014). Testing of Memory Leak in Android Applications. In *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*, (pp. 176–183). Miami Beach, FL, USA: IEEE.
- Soh, C., Tan, H. B. K., Arnatovich, Y. L., & Wang, L. (2015). Detecting Clones in Android Applications through Analyzing User Interfaces. In *2015 IEEE 23rd International Conference on Program Comprehension*, (pp. 163–173). Florence, Italy: IEEE.
- Sommerville, I. (1995). *Software Engineering*. Addison-Wesley, 5 ed.

- SourceLabs (2013). GTAC 2013: Appium: Automation for Mobile Apps. URL <https://www.youtube.com/watch?v=1J0aXDbjiUE> Accessed on 2016-07-07
- Standish, T. A. (1984). An Essay on Software Reuse. *IEEE Transactions on Software Engineering*, *SE-10*(5), 494–497.
- Su, T. (2016). FSMdroid: guided GUI testing of android apps. In *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE '16)*, (pp. 689–691). Austin, Texas, USA: ACM.
- Systems, S. (2012). Sparx Systems - UML 2 Tutorial - Sequence Diagram. URL http://www.sparxsystems.com/resources/uml2_tutorial/uml2_sequencediagram.html Accessed on 2012-07-12
- Takanen, A., DeMott, J., & Miller, C. (2008). *Fuzzing for Software Security Testing and Quality Assurance*. Norwood, MA, USA: Artech House, Inc., 1 ed.
- Utting, M., & Legeard, B. (2006). *Practical Model-Based Testing: A Tools Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers, 1 ed.
- van der Merwe, H., van der Merwe, B., & Visse, W. (2012). Verifying android applications using Java PathFinder. *ACM SIGSOFT Software Engineering Notes*, *37*(6), 1–5.
- van Rijsbergen, C. J. (1979). *Information Retrieval*. Newton MA, U.S.A.
- Vieira, V., Holl, K., & Hassel, M. (2015). A context simulator as testing support for mobile apps. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC '15)*, (pp. 535–541). Salamanca, Spain: ACM.
- Xamarin (2016). Calabash - automated acceptance testing for mobile apps. URL <http://calaba.sh/> Accessed on 2016-07-07
- Yang, S., Yan, D., Wu, H., Wang, Y., & Atanas Rountev (2015). Static control-flow analysis of user-driven callbacks in Android applications. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (Volume:1)*, (pp. 89–99). Florence, Italy: IEEE.
- Yang, W., Prasad, M. R., & Xie, T. (2013). 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)*, (pp. 250–265). Rome, Italy: Springer-Verlag.
- Yin, R. K. (2002). *Case Study Research: Design and Methods, 3rd Edition (Applied Social Research Methods, Vol. 5)*. SAGE Publications, Inc, 3rd ed.
- Yu, S., & Takada, S. (2015). External Event-Based Test Cases for Mobile Application. In *Proceedings of the Eighth International C* Conference on Computer Science & Software Engineering (C3S2E '15)*, (pp. 148–149). Yokohama, Japan: ACM.

- Zhou, W., Zhou, Y., Jiang, X., & Ning, P. (2012). Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy (CODASPY '12)*, (p. 317). San Antonio, Texas, USA: ACM.
- Zhu, H., Ye, X., Zhang, X., & Shen, K. (2015). A Context-Aware Approach for Dynamic GUI Testing of Android Applications. In *Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual*, vol. 2, (pp. 248–253). TaiChung, Taiwan: IEEE.

Appendix A

Results for the “Quality of the Results” Experiment

This Appendix presents the results of the experiment presented in subsection 5.5. For each application each of the three patterns (side drawer, orientation and tabs) was tested and thus for each application three tables are presented, one for each pattern. However, whenever a pattern is not present, no table is presented. This happens when: there is no side drawer in the application, the application does not respond to rotating the screen (the rotation is blocked for that application) or there are no tabs in the application.

Whenever a false positive or true negative occurs, the reason is analysed and described.

Books & References

JW Library

Table A.1: Results for the Side Drawer Pattern for the JW Library application

		iMPAcT	
		detects a failure	does not detect a failure
Manual inspection	is failure	1	0
	is not failure	0	-

Three of the false negatives are due to an error on the detection of a new activity and the remaining two are due to the time limit imposed for the execution (it does not reach this part of the application).

Table A.2: Results for the Orientation Pattern for the JW Library application

		iMPAcT	
		detects a failure	does not detect a failure
Manual inspection	is failure	19	5
	is not failure	2	-

The false positives are caused by the disappearance of some options from a context menu that in reality are moved into the Action Bar.

Table A.3: Results for the Tab Pattern for the JW Library application

		iMPAcT	
		detects a failure	does not detect a failure
Manual inspection	is failure	0	0
	is not failure	0	-

Scribd

It does not have a side drawer.

The orientation change is blocked for this application.

Table A.4: Results for the Tab Pattern for the Scribd application

		iMPAcT	
		detects a failure	does not detect a failure
Manual inspection	is failure	0	2
	is not failure	0	-

The false positives are caused by the same problem: one of the sets of tabs is not identified as such. Therefore, its bad positioning and the fact that it does not change the selected tab when swiping the screen are not detected.

Bible

Even though the side drawer does not visually occupy the full height screen, its root element does.

The false positive is caused by a problem in the scroll of UIAutomator.

Table A.5: Results for the Side Drawer Pattern for the Bible application

		iMPAcT	
		detects a failure	does not detect a failure
Manual inspection			
	is failure	0	1
	is not failure	0	-

Table A.6: Results for the Orientation Pattern for the Bible application

		iMPAcT	
		detects a failure	does not detect a failure
Manual inspection			
	is failure	6	5
	is not failure	1	-

One of the false negatives is due to iMPAcT not identifying a new activity as such, two of them occur due to the time limit imposed and the last two are due to UIAutomator not detecting the presence of two elements even though they are present.

Table A.7: Results for the Tab Pattern for the Bible application

		iMPAcT	
		detects a failure	does not detect a failure
Manual inspection			
	is failure	0	2
	is not failure	0	0

The two false negatives are caused by the misidentification of a set of tabs. As such, even though there are two sets of tabs in the same screen (there can only be one) and that the one that is not identified is on the bottom of the screen (if should be on the upper part), neither of these failures is detected.

Google Play Books

Table A.8: Results for the Side Drawer Pattern for the Google Play Books application

		iMPAcT	
		detects a failure	does not detect a failure
Manual inspection			
	is failure	1	0
	is not failure	0	-

Table A.9: Results for the Orientation Pattern for the Google Play Books application

	iMPAcT		
Manual inspection		detects a failure	does not detect a failure
	is failure	4	2
	is not failure	0	-

There are two false negatives because the failures are only detected after interacting with two elements to which it is not possible to access with the current exploration algorithm. The elements in question are hidden on a full screen application and are only visible if the user leaves the full screen mode (swiping the screen from the top to the bottom, for instance).

Table A.10: Results for the Tab Pattern for the Google Play Books application

	iMPAcT		
Manual inspection		detects a failure	does not detect a failure
	is failure	0	0
	is not failure	0	-

Wikipedia

Table A.11: Results for the Side Drawer Pattern for the Wikipedia application

	iMPAcT		
Manual inspection		detects a failure	does not detect a failure
	is failure	1	0
	is not failure	0	-

Table A.12: Results for the Orientation Pattern for the Wikipedia application

	iMPAcT		
Manual inspection		detects a failure	does not detect a failure
	is failure	0	0
	is not failure	0	-

It presents no set of tabs.

Business

Job Search

It does not have a side drawer.

Table A.13: Results for the Orientation Pattern for the Job Search application

		iMPAcT	
		detects a failure	does not detect a failure
Manual inspection	is failure	0	0
	is not failure	0	-

It presents no set of tabs.

File Commander

Table A.14: Results for the Side Drawer Pattern for the File Commander application

		iMPAcT	
		detects a failure	does not detect a failure
Manual inspection	is failure	0	0
	is not failure	0	-

Table A.15: Results for the Orientation Pattern for the File Commander application

		iMPAcT	
		detects a failure	does not detect a failure
Manual inspection	is failure	0	0
	is not failure	0	-

It presents no tabs.

Call Blocker Free

It presents no side drawer.

The orientation change is blocked for this application.

Table A.16: Results for the Tab Pattern for the Call Blocker Free application

	iMPAcT		
Manual inspection		detects a failure	does not detect a failure
	is failure	0	0
	is not failure	0	-

Table A.17: Results for the Orientation Pattern for the Trovit application

	iMPAcT		
Manual inspection		detects a failure	does not detect a failure
	is failure	5	1
	is not failure	0	-

Trovit

It presents no side drawer.

If the screen is rotated after checking an element, the *check* disappears. However, this happens because the *checked* property is not updated after the *check* event.

It presents no tabs.

BZ Reminder

Table A.18: Results for the Side Drawer Pattern for the BZ Reminder application

	iMPAcT		
Manual inspection		detects a failure	does not detect a failure
	is failure	0	1
	is not failure	0	-

Even though the side drawer does not visually occupy the full height screen, its root element does.

Table A.19: Results for the Orientation Pattern for the BZ Reminder application

	iMPAcT		
Manual inspection		detects a failure	does not detect a failure
	is failure	1	1
	is not failure	1	-

The false positive is due to the element disappearing from the context menu but appearing on the Action Bar in the landscape orientation.

The false negative occurs because the event that leads to the activity with the failure is never executed.

It presents no tabs.

Comics

Marvel’s Comics

Table A.20: Results for the Side Drawer Pattern for the Marvel’s Comics application

	iMPAcT		
		detects a failure	does not detect a failure
Manual inspection	is failure	0	1
	is not failure	0	-

In order for a side drawer to be identified as such it needs to be drawn on top of other elements. In this case, the UiAutomation considers the side drawer to be the only element in the screen and, thus, iMPAcT does not identify it as such and does not test it.

Table A.21: Results for the Orientation Pattern for the Marvel’s Comics application

	iMPAcT		
		detects a failure	does not detect a failure
Manual inspection	is failure	0	0
	is not failure	0	-

Table A.22: Results for the Tab Pattern for the Marvel’s Comics application

	iMPAcT		
		detects a failure	does not detect a failure
Manual inspection	is failure	0	0
	is not failure	0	-

Drawing Cartoons 2

The orientation change is blocked for this application.

Table A.23: Results for the Side Drawer Pattern for the Drawing Cartoons 2 application

	iMPAcT		
Manual inspection		detects a failure	does not detect a failure
	is failure	0	0
	is not failure	0	-

Table A.24: Results for the Tab Pattern for the Drawing Cartoons 2 application

	iMPAcT		
Manual inspection		detects a failure	does not detect a failure
	is failure	1	0
	is not failure	0	-

Draw Anime

Table A.25: Results for the Side Drawer Pattern for the Draw Anime application

	iMPAcT		
Manual inspection		detects a failure	does not detect a failure
	is failure	1	0
	is not failure	0	-

The orientation change is blocked for this application.

It presents no tabs.

DC Comics

Table A.26: Results for the Side Drawer Pattern for the DC Comics application

	iMPAcT		
Manual inspection		detects a failure	does not detect a failure
	is failure	0	1
	is not failure	0	-

In order for a side drawer to be identified as such it needs to be drawn on top of other elements. In this case, the UiAutomation considers the side drawer to be the only element in the screen and, thus, iMPAcT does not identify it as such and does not test it.

Table A.27: Results for the Orientation Pattern for the DC Comics application

		iMPAcT	
		detects a failure	does not detect a failure
Manual inspection		0	0
	is failure	0	0
	is not failure	0	-

Table A.28: Results for the Tab Pattern for the DC Comics application

		iMPAcT	
		detects a failure	does not detect a failure
Manual inspection		0	0
	is failure	0	0
	is not failure	0	-

Comics

Table A.29: Results for the Side Drawer Pattern for the Comics application

		iMPAcT	
		detects a failure	does not detect a failure
Manual inspection		0	1
	is failure	0	1
	is not failure	0	-

In order for a side drawer to be identified as such it needs to be drawn on top of other elements. In this case, the UiAutomation considers the side drawer to be the only element in the screen and, thus, iMPAcT does not identify it as such and does not test it. However, it is incorrectly drawn.

Table A.30: Results for the Orientation Pattern for the Comics application

		iMPAcT	
		detects a failure	does not detect a failure
Manual inspection		0	0
	is failure	0	0
	is not failure	1	-

The false positive is due to one of the options in the context menu being drawn in the Action Bar instead of in the context menu.

Table A.31: Results for the Tab Pattern for the Comics application

	iMPAcT		
Manual inspection		detects a failure	does not detect a failure
	is failure	0	0
	is not failure	0	-

Finance

Money Lover

Table A.32: Results for the Side Drawer Pattern for the Money Lover application

	iMPAcT		
Manual inspection		detects a failure	does not detect a failure
	is failure	0	0
	is not failure	0	-

Table A.33: Results for the Orientation Pattern for the Money Lover application

	iMPAcT		
Manual inspection		detects a failure	does not detect a failure
	is failure	15	5
	is not failure	0	-

Four of the false negatives are due to the time limit imposed and the remaining one is due to a new activity not being identified as such and, thus, not tested.

Table A.34: Results for the Tab Pattern for the Money Lover application

	iMPAcT		
Manual inspection		detects a failure	does not detect a failure
	is failure	0	0
	is not failure	0	-

Investing.com

In order for a side drawer to be identified as such it needs to be drawn on top of other elements. In this case, the UiAutomation considers the side drawer to be the only element in the screen and, thus, iMPAcT does not identify it as such and does not test it.

Table A.35: Results for the Side Drawer Pattern for the Investing.com application

		iMPAcT	
		detects a failure	does not detect a failure
Manual inspection		0	1
	is failure	0	1
	is not failure	0	-

The orientation change is blocked for this application.

Table A.36: Results for the Tab Pattern for the Investing.com application

		iMPAcT	
		detects a failure	does not detect a failure
Manual inspection		2	0
	is failure	2	0
	is not failure	0	-

Moneyfy

Table A.37: Results for the Side Drawer Pattern for the Moneyfy application

		iMPAcT	
		detects a failure	does not detect a failure
Manual inspection		1	0
	is failure	1	0
	is not failure	0	-

The orientation change is blocked for this application.

It presents no tabs.

MSN Money

Table A.38: Results for the Side Drawer Pattern for the MSN Money application

		iMPAcT	
		detects a failure	does not detect a failure
Manual inspection		0	1
	is failure	0	1
	is not failure	0	-

In order for a side drawer to be identified as such it needs to be drawn on top of other elements. In this case, the UiAutomation considers the side drawer to be the only element in the screen and, thus, iMPAcT does not identify it as such and does not test it.

The orientation changed is blocked in this application.

Table A.39: Results for the Tab Pattern for the MSN Money application

Manual inspection	iMPAcT		
		detects a failure	does not detect a failure
	is failure	0	0
is not failure	0	-	

Meta Trader 4

Table A.40: Results for the Side Drawer Pattern for the Meta Trader 4 application

Manual inspection	iMPAcT		
		detects a failure	does not detect a failure
	is failure	0	1
is not failure	0	-	

In order for a side drawer to be identified as such it needs to be drawn on top of other elements. In this case, the UiAutomation considers the side drawer to be the only element in the screen and, thus, iMPAcT does not identify it as such and does not test it.

Table A.41: Results for the Orientation Pattern for the Meta Trader 4 application

Manual inspection	iMPAcT		
		detects a failure	does not detect a failure
	is failure	3	2
is not failure	0	-	

The false negative is due to the time limit imposed.

It presents no tabs.

Table A.42: Results for the Side Drawer Pattern for the Clue - Period Tracker application

		iMPAcT	
		detects a failure	does not detect a failure
Manual inspection			
	is failure	0	0
	is not failure	0	-

Table A.43: Results for the Orientation Pattern for the Clue - Period Tracker application

		iMPAcT	
		detects a failure	does not detect a failure
Manual inspection			
	is failure	6	1
	is not failure	0	-

Health & Fitness

Clue - Period Tracker

The false negative is caused by the elements being programatically but not visually present in the screen.

Table A.44: Results for the Tab Pattern for the Clue - Period Tracker application

		iMPAcT	
		detects a failure	does not detect a failure
Manual inspection			
	is failure	0	0
	is not failure	0	-

Pedometer

It presents no side drawer.

The orientation changed is blocked for this application.

Table A.45: Results for the Tab Pattern for the Pedometer application

		iMPAcT	
		detects a failure	does not detect a failure
Manual inspection			
	is failure	0	1
	is not failure	0	-

One set of tabs was not identified and, thus, not tested.

7 Minute Workout

It presents no side drawer.

The orientation change is blocked for this application.

Table A.46: Results for the Tab Pattern for the 7 Minute Workout application

	iMPAcT		
		detects a failure	does not detect a failure
Manual inspection	is failure	0	0
	is not failure	0	-

Abs Workout

Table A.47: Results for the Side Drawer Pattern for the ABS Workout application

	iMPAcT		
		detects a failure	does not detect a failure
Manual inspection	is failure	0	0
	is not failure	0	-

The orientation change is blocked for this application.

It presents no tabs.

S Health

It presents no side drawer.

The orientation change is blocked for this application.

Table A.48: Results for the Tab Pattern for the S Health application

	iMPAcT		
		detects a failure	does not detect a failure
Manual inspection	is failure	0	0
	is not failure	0	-