2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop

Testing approach for mobile applications through reverse engineering of UI Patterns

Inês Coimbra Morgado
Department of Informatics Engineering
Faculty of Engineering of University of Porto
INESC TEC, Porto
Portugal
Email: pro11016@fe.up.pt

Ana C. R. Paiva
Department of Informatics Engineering
Faculty of Engineering of University of Porto
INESC TEC, Porto
Portugal
Email: apaiva@fe.up.pt

***Abstract*—It is increasingly important to assess and ensure the correct behaviour of mobile applications as their importance in everyday life keeps increasing. This paper presents an automatic testing approach combining reverse engineering with testing. The algorithm tries to identify existing User Interface (UI) patterns on a mobile application under test through a reverse engineering process and then tests them using generic test strategies called Test Patterns. The overall testing approach was implemented in the iMPAcT (Mobile PAttern Testing) tool and is illustrated in a case study performed over some mobile applications as a proof-of-concept.**

***Keywords*—Mobile Testing, GUI Testing, Pattern-based Testing, Reverse Engineering**

I. Introduction

Since the release of the iPhone in 2007 [11] and of the first Android smart phone in 2008 [7], [44], smart phones have started to greatly increase their sales. In fact, in 2013 both Android's Google Play and Apple's App Store surpassed the one million available applications and fifty billion downloads thresholds [26]. This market dimension makes it extremely important to ensure the quality of an application as it generates a high level of competitiveness and, thus, applications must be as flawless as possible in order to get popular. Furthermore, the number of business critical mobile applications like mobile banking applications is also increasing, which makes it even more important to ensure their functional correctness. However, due to peculiarities of the mobile world, such as their event-based nature, new development concepts like activities, new interaction gestures and limited memory, the mobile application testing process is a challenging activity [6], [35]. According to the World Quality Report 2014-15 [14], the number of organisations performing mobile testing is growing from 31% in 2012 to 55% in 2013 and near 87% in 2014. The same report mentions that the greatest challenge for mobile testing is the lack of the right testing processes and methods, followed by insufficient time to test and absence of in-house mobile test environments. As such, it is extremely important to automate mobile testing.

The automation of the test process can focus the automation of the execution of the test cases (Robotium[1], Selendroid[2] or Espresso[3]) and/or of the generation of the test cases. The first has several problems that need to be tackled, such as the variety of devices, which makes the execution of the test scripts a challenge, and the variety of platforms, which hardens the maintenance of the test scripts. A technique that enables the automation of test cases generation is Model Based Testing (MBT) [42], [28], [32], [45], [43], [12], [34], in which the test cases are generated from a specification/model.

The two main issues of MBT are 1) the necessity of an input model of the application, whose manual construction is a time consuming and error prone process and 2) the combinatorial explosion of test cases. The first problem may be tackled by increasing the level of abstraction of the models and/or by reverse engineering part of the application under test (AUT). The latter may be tackled by carefully selecting a subset of the final tests to execute and/or by focusing the tests on specific areas of the AUT, as it is the case of the Pattern Based GUI Testing project [33], which tests only recurring behaviour, also known as User Interface (UI) patterns, *i.e.*, recurring solutions for recurring problems.

There are some studies that show the usefulness of using patterns for mobile applications testing. In 2009, Erik Nilsson [38] identified some recurring problems when developing an Android application and the UI design patterns that could help solve them. In 2013, Sahami Shirazi *et al.* [41] studied the layout of Android applications trying, among other goals, to verify if these layouts presented any patterns. They concluded that 75.8% of unique combinations of elements appeared only once in the application. This study was conducted taking into consideration a static analysis of the layout and its elements. There is also some literature on the presence of patterns in mobile applications, such as [37].

As previously stated, software reverse engineering techniques are a valuable asset to the testing process. In 1990, Chikofsky and Cross [15] defined it as "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". As far as we know, none of the reverse engineering approaches applied on mobile applications tries to take advantage of the existence of UI Patterns on the application to facilitate the task.

[1] http://robotium.googlecode.com/
[2] http://selendroid.io/
[3] https://developer.android.com/training/testing/ui-testing/espresso-testing.html

978-1-4673-9775-9/15 $31.00 © 2015 IEEE
DOI 10.1109/ASEW.2015.11

This paper presents an approach for reverse engineering an Android mobile application and testing the UI Patterns it presents. This is achieved by defining a catalogue of patterns: a set of the UI Patterns and the corresponding test strategies to test them, *i.e.*, the corresponding Test Patterns. It works in a cycle which explores the mobile application through a reverse engineering process trying to identify the patterns. When a pattern is found, it is tested and the exploration continues. Section III-B explains this in detail.

The remaining of this paper is structured as follows. Section II presents related work in the fields of both mobile reverse engineering and mobile testing, with a special focus on the usage of patterns on mobile testing. Section III presents the approach followed and a sample of the patterns' catalogue. Section IV presents some preliminary results. Section V draws some conclusions.

II. Related Work

This Section presents some of the related work on mobile reverse engineering and mobile testing.

A. Mobile Reverse Engineering

Software reverse engineering is a field that has long been a field of research. There are several approaches that extract models from desktop [24], [39], [23], [18] and web [20], [3], [31], [30], [40] applications. However, in this Section only the research on reverse engineering of mobile applications will be taken into consideration. Moreover, Android reverse engineering will be the main focus for two reasons: first, the approach presented in this paper is applied to Android applications; second, the vast majority of approaches focuses on Android applications. In fact, to the best of our knowledge in the last years there is no approach focusing Windows Phone applications and only a handful of approaches dealing with iOS ones, such as Joorabchi *et al.* [27], who extract the model of an iOS application's UI, and Franke *et al.* [21], who extract the life-cycle of a mobile application, being it Android, iOS or JavaME.

When the focus of reverse engineering is to obtain the source code of the application from the APK (Android Application Package), *i.e.*, the Android executable file, Java disassembling or decompiling techniques may be used as long as the application is not obfuscated. As such, reverse engineering approaches usually focus on how to automatically analyse the source code, *i.e.*, static reverse engineering, or on how to automatically analyse the application at run time, *i.e.*, dynamic reverse engineering. There are also some approaches trying to bring together the best of both worlds by analysing both the source code and the application at run time, *i.e.*, hybrid reverse engineering. Taking into consideration the event-based nature of mobile applications, dynamic and hybrid approaches are the most common. Nevertheless, there are some like Batyuk *et al.* [13] who identify possible security vulnerabilities like unwanted user access that apply static approaches.

Regardless of the techniques used, the purpose of reverse engineering in the context of mobile applications is to obtain a model of the application and/or to test it. The work of Yang *et al.* [46] is an example of the first as it presents a hybrid approach: an initial static phase identifies the possible events to be fired and a second dynamic phase explores the application by firing those events and analysing their effects on the application. An example of the latter is the work of Amalfitano *et al.* in 2012 [6] which is similar to the exploration phase of the approach presented in this paper and in 2013 [4], in which they generate test cases but follow a dynamic approach with reflection and code replacement techniques.

B. Mobile Testing

Mobile applications testing (or mobile testing) has been gaining attention by researchers as it offers challenges that are different from the ones presented by web or desktop applications [35]. Testing may be manual, which will not be taken into consideration in this paper, or automated. Test automation can be focused on the generation or on the execution of test cases as stated in Section I.

The market already offers several options regarding the automation of the tests execution, including two official ones for Android: Espresso[4] and UIAutomator[5]. The main difference between these frameworks is that Espresso focuses on a single application while UIAutomator enables interacting with other applications besides the one under testing.

Considering the existence of official tools for automating the execution of the tests, researchers have been focusing on automating the generation of test cases. Nevertheless, most of them also execute the test cases after their generation in order to have a more complete approach.

Alike software testing, mobile testing can be divided in white-box testing, *i.e.*, accessing the application's source code, and black-box testing, *i.e.*, without accessing the source code. Gao *et al.* [22] provide an overview on mobile testing identifying four main testing purposes: UI testing, mobile connectivity testing, usability testing and mobility testing; and four main testing approaches: emulation-based testing, device-based testing, cloud testing and crowd based testing. They have also compared some mobile testing tools, like Appium[6] and MonkeyRunner[7].

In the academy, the interest in mobile testing has also increased. For instance, Amalfitano *et al.* [5] automates crash testing on Android applications, Machiry *et al.* [29] generates inputs to test the application, Costa *et al.* [19] generate test cases from a patttern-based model of the application, and Nasim *et al.* [36] identify malicious code automatically injected in the application's source code.

1) Patterns for Mobile Testing: Pattern identification is useful for testing mobile applications [38], [41]. However, sometimes different approaches focus on different types of patterns. For instance, Hu and Neamtiu [25] associate the notion of pattern with possible error classes, *i.e.*, typical places where errors can occur (activities, events); Batyuk *et al.* [13] define a pattern as a set of malicious access; Amalfitano *et al.* [4], [2] consider three types of patterns: GUI patterns, event

[4]https://developer.android.com/training/testing/ui-testing/espresso-testing.html

[5]https://developer.android.com/training/testing/ui-testing/uiautomator-testing.html

[6]http://appium.io/

[7]http://goo.gl/QaxnDJ

patterns and model patterns; and Costa *et al.* [19] consider UI Test Patterns.

Both Amalfitano *et al.* and Costa *et al.*'s patterns have some similarities with the ones presented in this paper. Amalfitano *et al.* define GUI and event patterns which represent the behaviour based on the GUI of the application and on Android system events, respectively. The UI Patterns presented in this paper can be related either to the GUI or to the system's events like detecting the presence of a side drawer or detecting if a certain resource is being used. Thus, Amalfitano *et al.*'s patterns are closely related to this paper's UI Patterns. The definition of Test Patterns used by Costa *et al.* is the same as the one used on this paper.

III. Overall testing approach

This approach has four main characteristics:

1) the reverse engineering process is fully dynamic, *i.e.*, the source code is never accessed and no code instrumentation is required;
2) the goal is to test recurring behaviours, *i.e.*, UI Patterns;
3) the whole process is completely automatic;
4) it is an iterative process combining reverse engineering and testing.

This Section is divided in two different sub-sections: description of the approach and description of the catalogue of patterns.

A. Patterns

Christopher Alexander first defined patterns in architecture as a representation of the "current best guess as to what arrangement of the physical environment will work to solve the problem presented" [1]. Generalising this concept, one can assert that a pattern is a recurring solution for a recurring problem.

In order to ensure the reuse of the patterns defined and to ease the process of defining new ones, it is important to formally define them. As such, a pattern is represented by the tuple <Goal, V, A, C, P>, in which:

Goal	is the ID of the pattern;
V	is a set of pairs {variable, value} relating input data with the variables involved;
A	is the sequence of actions to perform;
C	is the set of checks to perform;
P	is the precondition (boolean expression) defining the conditions in which the pattern should be applied.

In other words a pattern is represented as

$$G[configuration] : P- > A[V]- > C \quad (1)$$

, *i.e.*, for each goal's configuration G[Configuration], if the precondition (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 paper, two types of patterns are considered: UI Patterns and Test Patterns. A UI Pattern may be directly related either to the GUI of the application, such as the presence of the Action Bar, Side Drawer or Login/Password or system events such as rotating the screen, receiving an incoming call or losing wireless connectivity.

The presented pattern definition is applied both to the UI Patterns and to the Test Patterns. 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 its presence. On the other hand, in a Test Pattern, *P* defines when the test is applied (this must include the validation of the corresponding UI Pattern), *A* defines the actions of the test itself and *C* indicates if the test passed or failed, *i.e.*, it represents the oracle of the test.

B. The iMPAcT tool

This Section presents the approach, which is implemented in a tool called iMPAcT[8] (Mobile PAttern Testing). This approach (Figure 1) is divided in iterations of three phases each: exploring the application, identifying the existing UI Patterns and applying the corresponding test strategies (Test Patterns). Afterwards, the exploration continues.

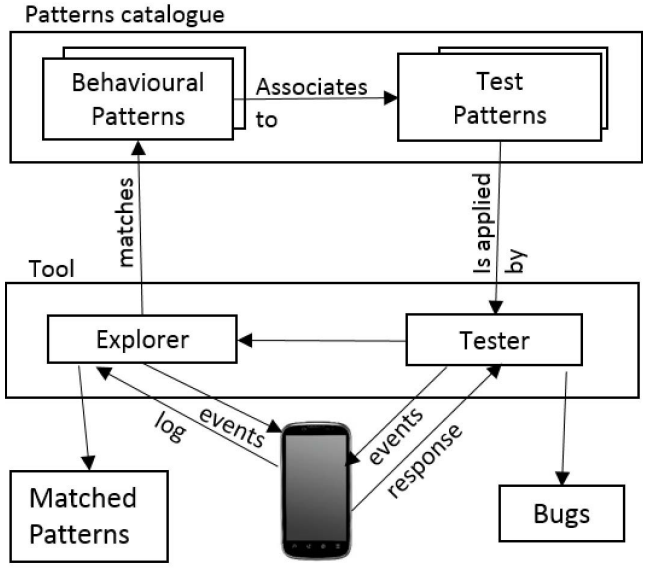

Fig. 1. Block Diagram of the Architecture of the Approach[16]

1) Exploration: The first phase of the iteration is the exploration. This consists on analysing the current state of the application, deciding on which event to fire, and firing it. The analysis of the current state consists on identifying the different elements (*e.g.*, buttons, text boxes) of the GUI in the current screen and determining which events are enabled in each of these elements (*e.g.*, click, edit, check). These events may vary for the same type of element. For instance, it is usually possible to click on a *button* but sometimes it is also possible to long-click it, *i.e.*, to press for a period of time instead of immediately

[8]A demo of the tool may be found in http://goo.gl/xx4W8O

lifting the finger. The decision of which event to fire can be random, alike a monkey testing process, or it may follow a heuristic. At the moment, it is random. The selected event is then fired.

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. All the UI Patterns of the catalogue are analysed in order to verify if they are currently found in the application.

Matching an UI Pattern consists on verifying if its *pre-condition* holds, executing any *actions* necessary and verifying if all *checks* are met. If so, the UI Pattern is considered found.

3) Testing: Whenever a pattern is found it must be tested. Each UI Pattern (UIP) has a Test Pattern (TP) associated. When an UIP is detected, the corresponding TP is applied. Applying a Test Pattern is similar to matching an UI one: verify if the *pre-condition* is met, execute the necessary *actions* and verify the *checks*. If everything goes smoothly, *i.e.*, if all *checks* are met the test passes. Otherwise, the test fails and a report is generated. If no pattern is found, this phase is skipped.

C. Patterns Catalogue

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 on a set of UI Patterns and the corresponding test patters, *i.e.*, the test strategies. A first draft of possible Test Patterns is described in [17]. However, in [17] only the Test Patterns were considered. This Section describes them as pairs UI Pattern - Test Pattern. Following, some patterns identified by the authors are presented. If more patterns should be identified and tested, they just need to be defined.

The patterns currently present in this catalogue are based on the guidelines provided by Android on how to design applications [10] and on how to test them [8].

1) Side Drawer Pattern: The Android OS provides several forms of navigation through its different screens and hierarchy. One of these is the *Side Drawer* (or Navigation Drawer) UI Pattern [9], [37], *i.e.*, a transient menu that opens when the user swipes the screen from the left edge to the centre or clicks on the application icon on the left of the application's *Action Bar*. Figure 2 depicts an example of this UI Pattern.

The UI Pattern is:

Goal: "Side Drawer exists"

V: {}

A: []

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

P: {true}

The corresponding Test Pattern is:

Goal: "Side Drawer occupies full height"

V: {}

A: [open side drawer]

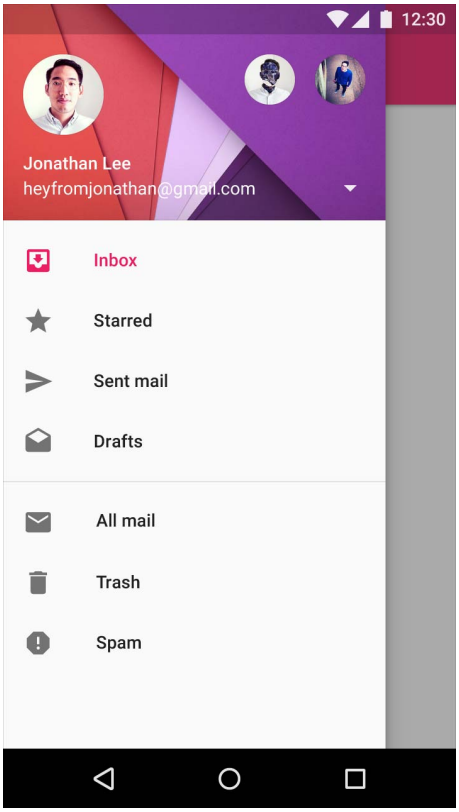

Fig. 2. Example of the side drawer pattern [9]

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

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

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

2) Orientation Pattern: Android devices have two possible orientations: portrait and landscape, as depicted in *a* and *b* of Figure 3, 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 [8] there are two main aspects the developers should test: no user input data should be lost, *i.e.*, all the content that the user has entered, such as text in *text edit*, or checked a *check box*, is still present after the rotation, and custom UI code can handle the changes, *i.e.*, ensure that the main components of the layout are still present even if not every element is present, *e.g.*, when rotating the screen a previously existing list must still be present even if not all its items are present due to lack of space. This is depicted in Figure 3 as the same lists are present in a) and b) even though the three elements of the second row of a) are not present in b).

The UI Pattern is:

Goal: "Rotation is possible"

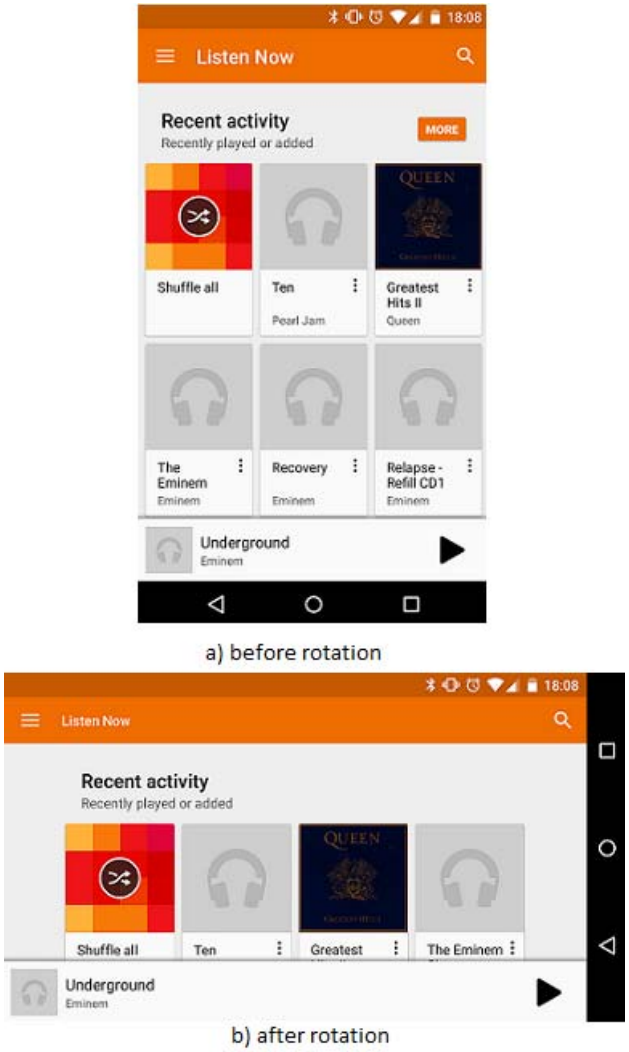

Fig. 3. Example of the possible orientations: a) portrait and b) landscape

V: {}

A: []

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

P: {"true"}

The corresponding Test Patterns are:

Goal: "Data unchanged when screen rotates"

V: {}

A: [read screen, rotate screen, read screen]

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

P: {"UIP is present && user data was entered && TP not applied to current activity"}

and

Goal: "UI main components are still present"

V: {}

A: [read screen, rotate screen, read screen]

C: {"main components still present"}

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

The main challenges of implementing this pattern are how to match the elements from the pre-rotation screen to the ones from the post-rotation screen as there is no unique id to identify them and which are the elements that must be present in both screen in the second Test Pattern.

3) Resources Dependency Pattern: Several 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 indicated by [8]. The corresponding UI Pattern is:

Goal: "Resource in use"

V: {"resource", resource_name}

A: []

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

P: {"true"}

The corresponding Test Pattern is:

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

V: {"resource", resource_name}

A: [read screen, turn resource off, read screen]

C: {"application did not crash"}

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

There can be multiple samples of this pattern in the catalogue, one for each resource to test. The resource may be, for instance, Wifi, 3G signal, GPS or Bluetooth.

The main challenge of implementing this pattern is how to verify if an application has crashed.

All these pattern share the challenge of how to differentiate activities and so it is necessary to take special care with this.

IV. Case Study

In order to validate the testing approach here presented a set of mobile applications was selected. This set is composed of applications used by other testing approaches and can be found in Tables I and II.

The iMPAcT tool was applied to these mobile applications and the results are reported in Table III. A cell may be marked with:

- 'NA' - Not Applicable, meaning that the Test Pattern was not executed. For instance, when the application under test does not have a side drawer, it does not make sense to apply the corresponding Test Pattern;

TABLE I. APPLICATIONS TESTED WITH THE IMPACT TOOL: FIRST PART

Applications	Size (MB)	Thousands of Downloads	Domain
Book Catalogue[9]	9.81	100-500	Productivity
TomDroid[10]	1.0	10-50	Productivity
Tippy Tipper[11]	3.12	100-150	Finance
Google Slides[12]	83.74	10,000-50,000	Productivity
Calendar[13]	33.25	100,000-500,000	Productivity

TABLE II. APPLICATIONS TESTED WITH THE IMPACT TOOL: SECOND PART

Applications	Reviews Classification	# of Reviews
Book Catalogue	4.4	275
TomDroid	3.9	718
Tippy Tipper	4.6	777
Google Slides	4.1	92,326
Calendar	4.0	404,469

- 'FF' - Failure Found, meaning that a failure was found in the UI Pattern;
- 'AF' - Absence of Failures, meaning that no failures were detected.

A cell with a The input value for the resource dependency pattern used was *Wifi*.

TABLE III. SUMMARY OF TEST RESULTS: FF-FAILURE FOUND; AF-ABSENCE OF FAILURES; NA-NOT APPLICABLE

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

The following failures were detected:

1) Book Catalogue: rotation is not correctly handled. When a list of options is displayed, if the screen is rotated the list disappears, *e.g.*, going to administrator preferences and selecting book list state;
2) Tippy Tipper: no failures were detected;
3) TomDroid: no failures were detected;
4) Google Slides: rotation is not correctly handled. For example, when opening "More options" (three vertical dots at the right of the Action Bar) rotating the screen makes this menu disappear;
5) Calendar: the side drawer does not fill in the full height of the screen;
6) Calendar: rotation is not handled correctly. For example: when selecting the month a calendar appears, rotating the screen makes it disappear; when opening the "Additional Menu" option (three vertical dots at the right of the Action Bar) rotating the screen makes it disappear. Rotating the screen at any moment makes the action bar disappear.

V. CONCLUSIONS

This paper presents an approach to test recurring behaviour in Android mobile applications, *UI Patterns*. This approach consists in exploring the application and testing if the present UI Patterns are correctly implemented. Previous approaches have proven the usefulness of defining test strategies for testing recurring behaviour [33] as well as the usefulness of using reverse engineering techniques to improve testing [5], [29], [40].

This approach presents a solution to two problems that usually occur in automated testing approaches: the necessity of defining what the correct behaviour is and the explosion of generated test cases. These problems are tackled by defining a catalogue of patterns containing which behaviours will be tested (UI Patterns) and how to test them (Test Patterns). The approach was implemented in a new tool called iMPAcT (Mobile PAttern Testing).

The conducted case study provides some insight into the usefulness of the approach. Even though it was applied on a small sample of mobile applications, it already showed that the iMPAcT tool is able to detect errors in the implementation, such as in the case of a change in orientation. Thus, it possible to conclude that this is a feasible and useful approach.

The tool still presents some limitations that need to be tackled in the future, namely:

1) improving the catalogue by adding more patterns as the quality of the test results is connected to the quality of the patterns;
2) adding different exploration approaches apart from random, *e.g.*, depth-first and guided; and
3) refinement of the exploration's stop condition.

Even though the iMPAcT tool is applicable on Android applications, the same approach could be used to test mobile application from other operation systems.

ACKNOWLEDGMENT

This work is financed by the ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness), POPH-QREN Programme (operational programme for human potential) and by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within the PhD scholarship SFRH/BD/81075/2011.

REFERENCES

[1] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, Berkeley, CA, USA, 1977.

[2] D. Amalfitano, N. Amatucci, A. R. Fasolino, U. Gentile, G. Mele, R. Nardone, V. Vittorini, and S. Marrone. Improving Code Coverage in Android Apps Testing by Exploiting Patterns and Automatic Test Case Generation. In *International workshop on Long-term industrial collaboration on software engineering (WISE 2014)*, pages 29–34, Västerás, Sweden, 2014. ACM.

[3] D. Amalfitano, A. R. Fasolino, and P. Tramontana. Experimenting a reverse engineering technique for modelling the behaviour of rich internet applications. In *2009 IEEE International Conference on Software Maintenance*, pages 571–574. IEEE, Sept. 2009.

[4] D. Amalfitano, A. R. Fasolino, P. Tramontana, and N. Amatucci. Considering Context Events in Event-Based Testing of Mobile Applications. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pages 126–133. IEEE, Mar. 2013.

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

[6] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using GUI ripping for automated testing of Android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*, page 258, New York, New York, USA, Sept. 2012. ACM Press.

[7] G. Android. Announcing the Android 1.0 SDK, release 1, Sept. 2008.

[8] G. Android. Android - What To Test, 2015.

[9] G. Android. Android Navigation Drawer, 2015.

[10] G. Android. Up and running with material design, 2015.

[11] Apple. Apple Reinvents the Phone with iPhone, 2007.

[12] S. Arlt, C. Bertolini, and M. Schäf. Behind the Scenes: An Approach to Incorporate Context in GUI Test Case Generation. In *IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2011)*, pages 222–231, Washington, DC, USA, 2011.

[13] L. Batyuk, M. Herpich, S. A. Camtepe, K. Raddatz, A.-D. Schmidt, and S. Albayrak. Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within Android applications. In *2011 6th International Conference on Malicious and Unwanted Software*, pages 66–72. IEEE, Oct. 2011.

[14] Capgemini, HP, and Sogeti. World Quality Report 2014-15, 2014.

[15] E. Chikofsky and J. Cross. Reverse Engineering and Design Recovery: a Taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[16] I. Coimbra Morgado, A. C. Paiva, and J. P. Faria. Automated Pattern-Based Testing of Mobile Applications. In *2014 9th International Conference on the Quality of Information and Communications Technology*, pages 294–299, Guimarães, Portugal, Sept. 2014. IEEE.

[17] I. Coimbra Morgado and A. C. R. Paiva. Test Patterns for Android Mobile Applications. In *20th European Conference on Pattern Languages of Programs (Europlop 2015)*, 2015.

[18] I. Coimbra Morgado, A. C. R. Paiva, and J. a. Pascoal Faria. Dynamic Reverse Engineering of Graphical User Interfaces. *International Journal On Advances in Software*, 5(3 and 4):224–236, 2012.

[19] P. Costa, A. C. R. Paiva, and M. Nabuco. Pattern Based GUI Testing for Mobile Applications. In *9th International Conference on the Quality of Information and Communications Technology (QUATIC 2014)*, pages 66–74, Guimarães, Portugal, Sept. 2014. IEEE.

[20] C. Di Francescomarino, A. Marchetto, and P. Tonella. Reverse Engineering of Business Processes exposed as Web Applications. In *2009 13th European Conference on Software Maintenance and Reengineering*, pages 139–148. IEEE, 2009.

[21] D. Franke, C. Elsemann, S. Kowalewski, and C. Weise. Reverse Engineering of Mobile Application Lifecycles. In *18th Working Conference on Reverse Engineering (WCRE '11)*, pages 283–292. IEEE, Oct. 2011.

[22] J. Gao, X. Bai, W.-T. Tsai, and T. Uehara. Mobile application testing - a Tutorial. *Computer*, 47(2):46–55, 2014.

[23] A. M. P. Grilo, A. C. R. Paiva, and J. P. Faria. Reverse engineering of GUI models for testing. In *The 5th Iberian Conference on Information Systems and Technologies (CISTI '10)*, number July, pages 1–6. IEEE, 2010.

[24] D. R. Hackner and A. M. Memon. Test case generator for GUITAR. In *Companion of the 13th international conference on Software engineering (ICSE Companion '08)*, ICSE Companion '08, page 959, New York, New York, USA, 2008. ACM Press.

[25] C. Hu and I. Neamtiu. Automating gui testing for android applications. In *The 6th International Workshop on Automation of software test (AST '11)*, page 7, New York, New York, USA, May 2011. ACM Press.

[26] N. Ingraham. Apple announces 1 million apps in the App Store, more than 1 billion songs played on iTunes radio, Oct. 2013.

[27] M. E. Joorabchi and A. Mesbah. Reverse Engineering iOS Mobile Applications. In *2012 19th Working Conference on Reverse Engineering*, pages 177–186. IEEE, Oct. 2012.

[28] A. Kervinen, M. Maunumaa, T. Pääkkönen, M. Katara, W. Grieskamp, and C. Weise. Model-Based Testing Through a GUI. In W. Grieskamp and C. Weise, editors, *5th International Workshop on Formal Approaches to Testing of Software (FATES 2005)*, volume 3997 of *Lecture Notes in Computer Science*, pages 16–31, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[29] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: an input generation system for Android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*, page 224, New York, New York, USA, Aug. 2013. ACM Press.

[30] Y. Maezawa, H. Washizaki, and S. Honiden. Extracting Interaction-Based Stateful Behavior in Rich Internet Applications. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 423–428. IEEE, Mar. 2012.

[31] A. Marchetto, P. Tonella, and F. Ricca. Under and Over Approximation of State Models Recovered for Ajax Applications. In *2010 14th European Conference on Software Maintenance and Reengineering*, pages 236–239. IEEE, Mar. 2010.

[32] A. M. Memon. *A comprehensive framework for testing graphical user interfaces*. PhD thesis, Jan. 2001.

[33] R. M. L. M. Moreira and A. C. R. Paiva. 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)*, pages 863–866, New York, New York, USA, Sept. 2014. ACM Press.

[34] R. M. L. M. Moreira, A. C. R. Paiva, and A. Memon. A pattern-based approach for GUI modeling and testing. In *24th IEEE International Symposium on Software Reliability Engineering (ISSRE 2013)*, pages 288–297, Passadena, CA, Nov. 2013. IEEE.

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

[36] F. Nasim, B. Aslam, W. Ahmed, and T. Naeem. Uncovering Self Code Modification in Android. In *First International Conference on Codes, Cryptology, and Information Security, C2SI 2015*, pages 297–313, Rabat, Morocco, 2015. Springer.

[37] T. Neil. *Mobile Design Pattern Gallery: UI Patterns for Smartphone Apps*. O'Reilly Media, Inc., Sebastopol, Canada, 2nd edition, 2014.

[38] E. G. Nilsson. Design patterns for user interface for mobile applications. *Advances in Engineering Software*, 40(12):1318–1328, Dec. 2009.

[39] A. Rohatgi, A. Hamou-Lhadj, and J. Rilling. An Approach for Mapping Features to Code Based on Static and Dynamic Analysis. In *2008 16th IEEE International Conference on Program Comprehension*, pages 236–241. IEEE, June 2008.

[40] C. Sacramento and A. C. R. Paiva. 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)*, pages 105–115, Guimarães, Portugal, Sept. 2014. IEEE.

[41] A. Sahami Shirazi, N. Henze, A. Schmidt, R. Goldberg, B. Schmidt, and H. Schmauder. Insights into layout patterns of mobile user interfaces by an automatic analysis of android apps. In *5th ACM SIGCHI symposium on Engineering interactive computing systems*, pages 275–284. ACM, June 2013.

[42] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 1 edition, 2006.

[43] M. Vieira, J. Leduc, B. Hasling, R. Subramanyan, and J. Kazmeier. Automation of GUI testing using a model-driven approach. In *2006 international workshop on Automation of software test (AST 2006)*, pages 9–14, New York, New York, USA, May 2006. ACM Press.

[44] M. Wilson. T-Mobile G1: Full Details of the HTC Dream Android Phone, Sept. 2008.

[45] Q. Xie. Developing cost-effective model-based techniques for GUI testing. In *28th international conference on Software engineering - (ICSE 2006)*, pages 997–1000, New York, New York, USA, May 2006. ACM Press.

[46] W. Yang, M. R. Prasad, and T. Xie. A Grey-Box Approach for Automated GUI-Model Generation of Mobile Applications. In *16th International Conference on Fundamental Approaches to Software Engineering (FASE'13)*, pages 250–265, Rome, Italy, 2013.