The iMPAcT Tool for Android Testing

INÊS COIMBRA MORGADO, Faculty of Engineering of the University of Porto, Portugal
ANA C. R. PAIVA, Faculty of Engineering of the University of Porto & INESC TEC, Portugal

This paper presents iMPAcT tool that tests recurring common behavior on Android mobile applications. The process followed combines exploration, reverse engineering and testing to automatically test Android mobile applications. The tool explores automatically the App by firing UI events. After each event fired, the tool checks if there are UI patterns present using a reverse engineering process. If a UI pattern is present, the tool runs the corresponding testing strategy (Test Pattern). During reverse engineering the tool uses a catalog of UI Patterns which describes recurring behavior (UI Patterns) to test and the corresponding test strategies (Test Patterns). This catalog may be extended in the future as needed (*e.g.*, to deal with new interaction trends). This paper describes the implementation details of the iMPAcT tool, the catalog of patterns used, the outputs produced by the tool and the results of experiments performed in order to evaluate the overall testing approach. These results show that the overall testing approach is capable of finding failures on existing Android mobile applications.

CCS Concepts: • **Software and its engineering**; • **Software creation and management**; • **Software verification and validation**;

Keywords: Mobile testing; Android testing; Software testing; Software test automation; Reverse engineering; UI Patterns; Mobile crawler

ACM Reference Format:
Inês Coimbra Morgado and Ana C. R. Paiva. 2019. The iMPAcT Tool for Android Testing. In *Proceedings of the ACM on Human-Computer Interaction*, Vol. 3, EICS, Article 4 (June 2019). ACM, New York, NY. 23 pages. https://doi.org/10.1145/3300963

1 INTRODUCTION

Since the release of the iPhone in 2007 [15] and of the first Android smart phone in 2008 [9, 70], 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 threshold of one million available applications and fifty billion downloads [43]. The dimension of this market generated a high level of competitiveness where it is extremely important to ensure the quality of an application that must be as flawless as possible to become 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, peculiarities of mobile applications, such as their event-based nature, new development concepts like activities, new interaction gestures and limited memory, make the testing process of mobile applications a challenging activity [6, 56].

Authors' addresses: Inês Coimbra Morgado, Faculty of Engineering of the University of Porto , Rua Dr. Roberto Frias, s/n , Porto , 4200–465 , Portugal, ines.cmb.m@gmail.com; Ana C. R. Paiva, Faculty of Engineering of the University of Porto & INESC TEC , Porto , Portugal , apaiva@fe.up.pt.

2573-0142/2019/6-ART4 $15.00
https://doi.org/10.1145/3300963

According to the World Quality Reports 2014-15 [22] and 2015-16 [21], the number of organizations performing mobile testing is growing from 31% in 2012 to 55% in 2013, near 87% in 2014 and 92% in 2015. Both reports mention that the greatest challenge for mobile testing 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. Moreover, the 2015-16 report states that in 2015 there was a 9% increase in the amount of IT budget allocated to QA and Testing. As such, it is extremely important to automate mobile testing.

The test automation process can be focused on the test case execution and/or on the test cases generation [58]. In order to execute test cases automatically there are several problems that need to be tackled, such as dealing with the huge variety of devices and platforms, which hardens the maintenance of the test scripts. A technique that enables the automation of test cases generation is Model Based Testing (MBT) [16, 47, 55, 68, 71], in which the test cases are generated from a specification/model.

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, which makes test execution infeasible. The first problem may be tackled by increasing the level of abstraction of the models and/or by obtaining part of such model by a reverse engineering process 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. This is the case, for instance, of the Pattern Based GUI Testing project (PBGT) [31, 51, 53–55], which aims to test recurring behavior, also known as User Interface (UI) patterns.

Christopher Alexander [3] 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". Generalizing this concept, one can assert that a pattern is a recurring solution for a recurring problem.

There are some studies that show the usefulness of using patterns to test mobile applications. In 2009, Erik Nilsson [61] 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.* [66] studied the layout of Android applications trying, among others, to check if those layouts contained 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 [59].

As previously stated, software reverse engineering techniques may be a valuable asset to the MBT process since they may obtain (part of) the model from the application to test [2, 25, 62, 72]. In 1990, Chikofsky and Cross [23] defined it as "the process of analyzing 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".

This paper presents an approach and tool (iMPAcT) to automate the testing of mobile applications. It is an iterative process that combines reverse engineering and testing. It automatically explores the mobile application while trying to identify recurring behavior (pattern matching) to test. The whole process is based on a catalog of patterns that defines which type of behavior should be searched (UI Patterns) and the test strategies that can be applied in order to ensure the behavior is correctly implemented (Test Patterns).

This paper extends the research work presented in [25] by improving some sections and by describing the testing tool in more detail. First of all, the implementation of the exploration algorithm was improved with some heuristics in order to fire more events on the AUT. For instance, the up button is only pressed when there are no more events to fire on the current screen. Secondly,

a new UI Pattern is introduced (TabScroll), along with the corresponding Test Patterns, and the case study is extended accordingly. Thirdly, the visualization of the iMPAcT tool outputs were improved, namely by providing a state machine representing the different screens traversed during the exploration and the events that move along them. Finally, this paper adds a discussion comparing the iMPAcT tool with other approaches.

The remaining of this paper is structured as follows: Section 2 presents related work about mobile test automation and mobile reverse engineering; Section 3 presents the testing approach followed, including details on the implementation of the iMPAcT tool; Section 4 describes the catalog of patterns currently tested by the iMPAcT tool; Section 5 presents how the tool results are visualized; Section 6 presents some preliminary results; Section 7 discusses the results obtained and compares the iMPAcT tool with other approaches; and Section 8 draws some conclusions.

2 RELATED WORK

The work presented in this paper focuses on two main research fields: mobile test automation and mobile reverse engineering.

2.1 Mobile Test Automation

Mobile application testing (henceforth mobile testing) has been gaining interest by researchers because it is necessary to study either how to adapt the web and desktop testing approaches to mobile applications or how to define new approaches as Muccini *et al.* concluded in their study on the challenges and future research directions of mobile testing in 2012 [56].

From the eight main purposes for Verification and Validation (V&V) identified in the System and software Quality Requirements and Evaluation ISO [44], test automation approaches focus mostly on 1) functional testing, *i.e.*, verifying “the degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions" [30, 37, 60, 69], 2) reliability, *i.e.*, “degree to which a system, product or component performs specified functions under specified conditions for a specified period of time” [35, 67], 3) 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” [17] and 4) maintainability, *i.e.*, “degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers" [1, 73].

When automating mobile testing 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 alleged behavior of the AUT (*e.g.*, Nguyen *et al.* [60]), the correct life cycle an application should follow (*e.g.*, Franke *et al.* [35]), the correct filtering of intents (*e.g.*, Avancini and Ceccato [17]) or even information on how to classify the purpose of the application according to its description (*e.g.*, Gorla *et al.* [37]). 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.

The market also offers some tools that can ease the generation of test cases, such as the Monkey tool[1], which generates random sequences of events, and that can ease the execution of test cases. In fact, Google offers two official frameworks: Espresso[2], which is a single application testing framework launched on December 2014, and UIAutomator[3], which is a cross-application testing framework whose first version was launched on November 2012 and latest version was launched

[1]http://developer.android.com/intl/es/tools/help/monkey.html
[2]http://developer.android.com/training/testing/ui-testing/espresso-testing.html
[3]http://developer.android.com/training/testing/ui-testing/uiautomator-testing.html

on March 2015. Before these frameworks were available, the most common one was Robotium[4], launched in January 2010, which offered a simple testing API that worked on any device regardless of its Android version and, thus, rapidly spread within the community.

2.2 Mobile Reverse Engineering for Testing

As stated in Section 1, software reverse engineering aims at obtaining an abstraction of the application under analysis by identifying its components and interrelationships. There are several approaches that target desktop [28, 29, 38, 39, 64] and web [5, 48, 50, 57, 65] applications. However, we only consider research on reverse engineering of mobile applications. Moreover, only approaches targeting Android applications are considered because this is the operating system targeted by our approach and only a small number of researchers focus on other operating systems (*e.g.*, Joorabchi *et al.* [46], who targets iOS applications and Franke *et al.* [34], who targets Android, iOS and JavaME).

The majority of the 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.*, Verification and Validation (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 [46], who infer a model of the application's UI states stating they intend to use it to smoke test iOS applications, and Yang *et al.* [72], who obtain a model of the application's behavior with intentions of using it as the base for a MBT approach. Hence, this Section mainly considers approaches whose main goal is to test the application under analysis.

There are mainly three types of reverse engineering: static, which solely analyses the source code (or byte code) of the application; dynamic, which only considers the runtime information of the application; and hybrid, which takes advantage of the information extracted from both the source code and the runtime execution information. There are some works that statically analyze mobile applications but they usually focus security issues [18, 32, 63]. The vast majority of approaches either applies a hybrid approach, such as Hu *et al.* [41], who find and classify bugs in Android applications or a dynamic approach. However, most approaches that analyze the application during runtime, either run the application on an instrumented version of the emulator like the Dalvik VM or use an instrumented version of the AUT. Instrumenting the system means that a device/normal emulator can not be used and instrumenting the AUT means the application needs to be pre-processed before the testing process. In both cases, the runtime execution of the application is affected. Moreover, even when the instrumentation of the AUT is made automatically, maintaining an instrumented code is a difficult chore.

The main limitation of the approaches using reverse engineering is the lack of prior knowledge about the AUT. This restricts the aspects of the AUT that can be tested and it is the reason why no approach focuses functional sustainability testing and why most approaches focus on detecting crashes or unwanted accesses. Using the classification of the System and software Quality Requirements and Evaluation ISO, most approaches focus on reliability [6, 41, 42], security [18, 32, 49] and maintainability [8, 36, 45].

2.3 Pattern-based Testing

The simplest definition of pattern is a recurring solution for a recurring problem.

[4] http://robotium.com/

One of the most well-known patterns found on applications' UIs is the login/password. In this case the goal is to identify who is trying to access private parts of the application without the proper authorization.

As stated in 1, there are studies about the presence of patterns in mobile applications [61, 66] and Theresa Neil presented a compilation of some of these patterns [59].

Considering this, some approaches try to take advantage of their presence in order to ease the testing of mobile applications. The main difference between the different approaches is the type of patterns considered. Even though approaches like Batyuk *et al.*'s [18] and Shahriar *et al.*'s [67] focus on malicious intents and memory leaks, respectively, most patterns relate to the UI of the applications [4, 30, 40, 41, 73] and, thus, there are some similarities between them. Amalfitano *et al.*'s GUI and event patterns [4] can be compared to the UI Patterns presented by Costa *et al.* [30] as they represent behaviour commonly observed on mobile applications and Costa *et al.*'s UI Test Patterns can be compared to Holl and Elberzerger's patterns [40] as they illustrate how a failure on the AUT can be detected. Even though Hu and Neamtiu [41] focus on detecting failures, they are more focused on the faults that originated them. Yu and Takada [73], on the other hand, just classify the type of events that can be fired.

The approach presented in this paper applies a dynamic reverse engineering approach that neither requires instrumenting the source code of the AUT nor uses an instrumented version of the emulator. Therefore, it does not interfere with the behavior of the application nor does it involve any pre-processing. By taking advantage of the presence of the UI patterns and focusing the testing on these components, the approach is able to detect failures other than crashes. Hence, it is a functional sustainability testing approach.

3 OVERALL TESTING APPROACH

The testing approach presented in this paper has four main characteristics and is supported by the iMPAcT (Mobile PAttern Testing) tool:

(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 behavior, *i.e.*, UI Patterns;
(3) the whole process is completely automatic;
(4) it is an iterative process combining automatic exploration, reverse engineering and testing.

The iMPacT tool aims at testing recurring behavior, the so called UI Patterns. The tool is able to identify and test the UI Patterns that belong to a catalog (presented in Section 4). However, the tool is built so that it is possible to extend the catalog as needed (for instance, to include new interaction trends).

3.1 Patterns

The patterns considered in this approach are formally defined inside a catalog. Hence, 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.

ALGORITHM 1: iMPAcT tool execution algorithm

```
exploring = true;
while (exploring) do
    call fire_event;
    call find_UI_patterns;
    if (UI_pattern_is_found) then
        call apply_test_pattern;
    end
    read exploring;
end
```

In other words a pattern is represented as

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

, *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 paper, two types of patterns are considered: UI Patterns and Test Patterns. A UI Pattern may be directly related to the application GUI (such as Action Bar, Side Drawer, Login/Password), or to system events (such as rotating the screen, receiving an incoming call or losing wireless connectivity). A Test Pattern is a generic test strategy that is able to test the behavior of the corresponding UI Pattern.

The same formalization of a pattern is used for both the UI Patterns and the Test Patterns. However the meaning of the items have slightly different interpretations in each case. 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 (which includes checking if the corresponding UI Pattern exists), *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 final assertions that indicate if the test passes or fails.

3.2 The iMPAcT tool

The testing approach presented in this paper is supported by the iMPAcT tool[5] [26], which automates the testing of the recurring behavior (UI patterns) present on Android mobile applications. Both the tool and the patterns are implemented in Java. The tool was developed based on the Google's API UiAutomation[6] to read the screen of the device and Google's API UI Automator to interact with the device. The iMPAcT tool works in iterations of three phases: the iMPAcT tool continuously explores the AUT (*Explorer*) while trying to identify the presence of UI Patterns (*find_UI_patterns*), then it tests them (*Tester*), *i.e.*, applies the associated Test Pattern. The iMPAcT tool presents different outputs: the matched patterns, *i.e.*, which UI Patterns are present in the application, the failures, *i.e.*, which of those patterns are not correctly implemented in the AUT and a Finite State Machine (FSM) representing the screens of the AUT and how to navigate through them.

Algorithm 1 shows the general implementation of the approach.

The *fire_event* method executes events on the AUT, *find_UI_patterns* tries to detect the presence of UI Patterns and finally *apply_test_pattern* executes the corresponding test strategy (Test Pattern)

[5] //web.fe.up.pt/ apaiva/pbgtwiki/doku.php?id=impact

[6] https://developer.android.com/reference/android/app/UiAutomation.html

ALGORITHM 2: Function get_current_screen()

```
child = screen_root.first_child;
while (child exists) do
    add child to parent;
    child = child_of_the_child;
end
```

ALGORITHM 3: Function node.get_possible_events()

```
if node_is_editable then
    add Edit_event to the node;
end
if node_is_checkable then
    add check_event to node;
end
```

when an UI Pattern is found. Further details on each of these methods are presented ahead in this Section. Each Pattern contains a *goBack()* method that attempts to bring the application back to the point it was before the test took place. However, this may not be possible, because the test itself can change the state of the app being tested and moving back does not guaranty the app will be in the same state as before the test. For example, if the test consisted in changing the screen orientation the *goBack()* method would consist in rotating the screen back to the original orientation. If this test provokes the disappearance of a pop-up (a dialogue), the state after the test will not be the same as before the test.

3.2.1 Exploration. The exploration analyses the current state of the application and decides which event to fire.

A state of the application is defined by the hierarchy of the elements present in the current screen, *i.e*, a tree representing the layout of the screen in which each node (child in Algorithm 2) is an element of the screen. Further explanation on state definition is on Section 5. The elements that are visible on the screen (*e.g*., buttons, check boxes) are designated as widgets. Algorithm 2 implements the construction of this tree.

Each of these nodes may be associated with executable events, *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 lifting the finger. The identification of the events enabled for an element of the screen (a node of the tree) is implemented in Algorithm 3.

Currently, the events fired by iMPAcT tool that mimic user interactions are: “click”, “long click”, “check” and “edit”.

The other events implemented by the tool (namely, “scroll”, “swipe” and “rotate screen”) are used only during the testing phase, not fired during the exploration phase and so, their goal is not to mimic user actions.

In previous work [25], the decision of which event to execute was completely random. However this process was improved in order to allow firing more events. Currently, when identifying the events that can be executed on a certain screen (*get_possible_events* in algorithm 4), the tool prioritizes:

(1) not yet executed events that belong to a list;

ALGORITHM 4: Function fire_event()

```
call get_current_screen;
forall node in screen do
    call node.get_possible_events
end
call choose_event;
call execute_event;
```

ALGORITHM 5: Functions: 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;
    end
else
    return false;
end
```

(2) not yet executed events except clicking on the *Up Button* of the action bar which takes the AUT back to a previous state;
(3) events that were already executed but that lead to a screen that still has events to be executed;
(4) click on the *Up Button*.

If there are no events available the *back* button of the device is pressed.

The exploration phase can be summarized in Algorithm 4, which implements the *fire_event*() method of Algorithm 1.

When the exploration gets to the home screen of the device, the exploration ends, *i.e.*, the stop condition of the exploration process is reaching the home screen which can be forced by the tester by pressing the *home* button when desired .

3.2.2 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 Algorithm 1.

Matching an UI Pattern consists on 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.

3.2.3 Testing. Whenever an UI Pattern is detected, the corresponding Test Patterns are applied: verify if the *precondition* 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 true, the test passes. Otherwise, the test fails. At the end a report is generated. This corresponds to the *apply_test_pattern()* method of Algorithm 1. If no UI Pattern is found in the pattern matching phase, this phase is skipped.

Both finding an UI Pattern and applying a Test Pattern follow Algorithm 5.

4 PATTERNS CATALOG

One of the most crucial aspects of this approach is the catalog of patterns as it defines what will be tested and how. This catalog consists on a set of UI Patterns and the corresponding Test Patterns, *i.e.*, the test strategies. A first draft of Test Patterns is described in [24].

This Section describes the pairs (UI Pattern - Test Patterns) currently implemented on the iMPAcT tool. These patterns are based on the guidelines provided by Android on how to design applications [12] and on how to test them [10]. The patterns presented in this Section are a subset of the many patterns that may be encountered in mobile applications. According to the guidelines provided by Android [7], they can be classified belonging to the categories "Visual design and user interaction" and "Functionality". At this point in time, we are not covering "Compatibility, performance and stability", neither "security".

4.1 Side Drawer Pattern

The Android OS provides several forms of navigation through different screens and hierarchy. One of these is the *Side Drawer* (or Navigation Drawer) UI Pattern [11, 59], *i.e.*, a transient menu that opens when the user swipes the screen from the left edge to the center or clicks on the App button on the left of the application's *Action Bar*. Figure 1 depicts an example of this UI Pattern.

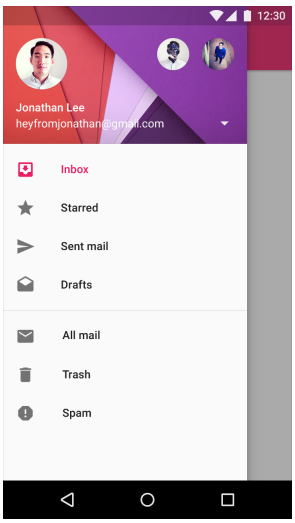

Fig. 1. Example of the side drawer pattern [11]

The UI Pattern that identifies the presence of a Side Drawer in a screen is defined as:

Goal: "Side Drawer exists"
V: {}
A: [read screen]
C: {"side drawer exists and is hidden"}
P: {true}

The corresponding test strategy (Test Pattern) that checks if the Side Drawer UI Pattern is correctly implemented, *i.e.*, if it takes up the full height of the screen is defined as:

[7] //developer.android.com/docs/quality-guidelines/core-app-quality

Goal: “Side Drawer takes up full height"
V: {}
A: [read screen, open side drawer, read screen]
C: {“takes up the full height of the screen"}
P: {“UIP present && side drawer available && TP not applied on current activity"}

The main challenge of implementing this pattern is on how to identify the side drawer element. This 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;
- screen must not contain a pop-up;
- the hierarchy of elements follow one of these aspects: a) the element is an only child, is of the type *listView* or *frameLayout*, its parent is a view and its grandparent is a *frameLayout*; b) the element is not an only child and is either a *frameLayout* or a *linearLayout*.

The items of the previous list were inferred by analyzing the structure of the side drawer in different mobile applications. All these applications were available in the Google Play Store and presented a Side Drawer.

4.2 Orientation Pattern

Android devices have two possible orientations: portrait and landscape, as depicted in *a* and *b* of Figure 2, respectively.

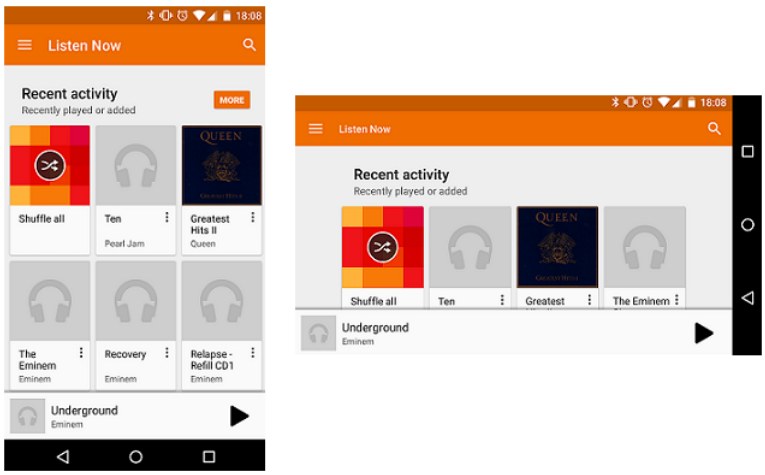

Fig. 2. Example of the possible orientations: a) portrait and b) landscape

When rotating the device, the screen of the application also rotates and its layout is updated. However, according to Android’s Guidelines for testing [10] 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 an *edit text*, or check on a *check box*, is still present after the rotation, and widgets should not disappear when rotating the screen. Figure 2 depicts a rotation of the screen in which some widgets present in the first screen (a) disappear from the the second one (b). However, this is not a failure as the items were simply dislocated due to lack of space. In order to avoid defining this situation as a failure, after a rotation, the screen is scrolled and the new elements are added to the screen's internal representation. The "scroll screen" action is needed to allow access to the full tree of GUI elements on the screen, in addition to those that are visible before scrolling.

The UI Pattern Orientation is defined as:

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

The corresponding Test Patterns are defined as:

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

and

Goal: "UI main components are still present"
V: {}
A: [read screen, rotate screen, read screen, scroll screen, read screen]
C: {"main components still present"}
P: {"UIP is present && TP not applied on current activity"}

The comparison of the before and after rotation screens reports a failure when one of the following situations occurs (these situations were inferred by analyzing different mobile applications from the Google Play Store):

(1) only one of the screens is a pop-up;
(2) only one of the screens has a side drawer;
(3) a widget is present on the first screen but not on the second one.

If a side drawer is detected in both screens, only the widgets contained within it are compared.

The main challenge of implementing this pattern is the match between the widgets from the first screen (before rotation) with the ones from the second screen (after the rotation). This is difficult because widgets do not have a unique ID. This match is achieved by comparing all the properties of the widgets except their position on the screen and user inserted data.

However this does not completely solve the problem as there are some widgets that are only distinguishable by their position, *e.g.*, check boxes within a list. Thus, in order to compare this type of widgets the text next to them is identified and compared.

4.3 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 [10].

The UI Pattern is defined as:

Goal: "Resource in use"

V: {“resource", resource_name}
A: [read resource status]
C: {“resource is being used by the app"}
P: {“true"}

The corresponding Test Pattern is defined as:

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 on current activity"}

Examples of values for the *resource_name* are wifi, 3G signal, GPS and Bluetooth.

The main challenge in implementing this pattern is to check whether the application crashed. This is achieved by verifying if a screen has both the following characteristics:

- it is a pop-up;
- its package name is “android".

4.4 Tab-Scroll Pattern

In some applications it is possible to find tabs, such as Facebook, Twitter or Skype. A tab (Figure 3) “makes it easy to explore and switch between different views or functional aspects of an application or to browse categorized data sets" [14].

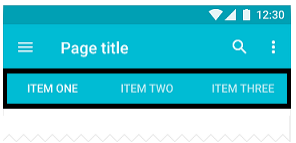

Fig. 3. Example of a tab

There are some design guidelines regarding tabs [19, 20] 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) swiping the screen horizontally should only change the selected tab, *i.e.*, there should be no widget enabling horizontal scroll apart from the one handling the tabs; and 3) as this catalog concerns Android applications, tabs should be located on the upper part of the screen (“Android’s tabs for view control are shown in action bars at the top of the screen" [13]) unlike what happens in iOS applications (“A tab bar always appears at the bottom edge of the screen"[33]).

The corresponding UI Pattern is defined as:

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

The corresponding Test Patterns are defined as:

Goal: “Only one set of patterns"
V: {}
A: [read screen]
C: {“there is only one set of tabs at the same time"}

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

and

Goal: “Horizontally scrolling a widget should change the selected tab"
V: {}
A: [read screen, swipe scrollable widget horizontally, read screen]
C: {“the selected tab changed"}
P: {“UIP && TP not applied on current activity"}

and

Goal: “Tabs position"
V: {}
A: [read screen]
C: {“Tabs are on the upper part of the screen"}
P: {“UIP && TP not applied on current activity"}

The main challenge in implementing this pattern is to identify whether an item is horizontally scrollable as the properties provided by UiAutomation only indicate whether or not it is scrollable. This is achieved by trying to horizontally scroll each of the scrollable widgets present in the screen and verifying if the tab selection changed.

All the patterns presented so far share the challenge of how to detect if a Test Pattern was already applied to a screen, *i.e.*, how to differentiate them. This is important so that the testing approach does not apply the same pattern more times than necessary thus saving time. This is described in more detail in Section 5.

5 VISUALIZATION OF THE RESULTS

In [25], the results obtained by the iMPAcT tool consisted of the execution trace with information on the matched UI Patterns and the detected failures (referred to as bugs in [25]). Throughout the experiments performed with the tool it was concluded that these reports could be improved.

Currently, the final report provides, apart from the logs, a graphical visualization of the exploration performed by the iMPAcT tool as a Finite State Machine (FSM), in which states are activities (an unique screen) of the application and transitions (arrows) are events fired. Hence, an arrow connecting state A with state B represents the event that when fired in state A causes a change to the state B of the application. The arrows are labeled with numbers representing events fired by the tool. An event has an id and information about the element in which it has occurred.

At this moment, the FSM generated is useful for reproducing the failures found. However, it may also be used in the future to prove properties about the mobile applications using model checking.

The UI Automator does not provide information about when the application reaches a new activity. So, an heuristic is used to identify activities: two screens are considered different if they do not have any widgets in common. The widgets that belong to an action bar (example in Figure 4) of an application are exceptions. These widgets are not considered in this set as it is expected that the same action bar, or some of its widgets, are present in different screens of the application, such as the search button or the more options button.

Fig. 4. Action Bar of the Google Slides application

Figure 5 depicts the state machine obtained by exploring the Tippy Tipper[8] application. The states are screen shots taken whenever a new screen is detected during the exploration. Screen 0 is the initial state of the application.

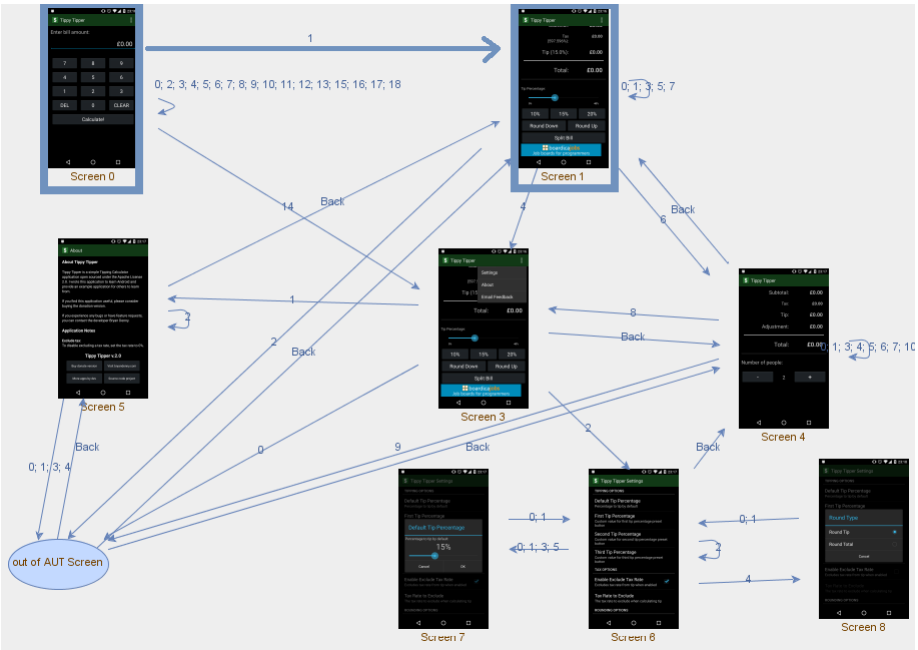

Fig. 5. State Machine of the Tippy Tipper application

Apart from the nodes representing the screens of the application, there may be two extra screens: the *out of AUT Screen*, which indicates that the exploration left the application, and the *crash Screen*, which indicates that a crash occurred. When an *out of AUT Screen* is detected, the *press back* event is executed in order to go back to the application and continue the exploration. Such a screen is depicted in the lower left part of the FSM of Figure 5. On the other hand, if an application crashed, the exploration continues by pressing the *Ok button*, which is always present in a crash pop-up. When clicking on an arrow of the FSM, further details on the events it represents are described.

This FSM is extremely useful for comprehending the behavior of the application, depicting its different states and how one can navigate through the application, and for easing the reproduction of the failures detected. For example, the path that leads to a failure (in the Orientation Pattern) is highlighted in bold and may be useful to reproduce the failures detected. In this case, in Screen 0 (the initial state) the user clicks on the "Calculate!" button going to Screen 1. If the screen is rotated the "Round Down" and "Round Up" buttons disappear.

6 CASE STUDY

In order to validate the overall approach we performed a case study. The goal was to answer the following research questions:

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

[8]https://play.google.com/store/apps/details?id=net.mandaria.tippytipper&hl=en

RQ2: Does the iMPAcT tool performance depend on the size of the applications? This can be divided into two sub-questions: a) is the coverage acceptable, *i.e.*, is it able to explore the majority of the identified events? b) does the whole process take a reasonably amount of time?

For this case study, we selected a set of mobile applications that are either used by other researchers during their validation process [4, 30, 72] or are official Google applications. This set can be seen in Table 1.

Table 1. Applications tested with the iMPAcT tool

Applications	Size (MB)	Downloads (x1000)	Domain	Classification	Reviews
Book Catalog	9.81	100-500	Productivity	4.4	3121
Tomdroid	1.0	10-50	Productivity	3.9	737
Tippy Tipper	3.12	100-150	Finance	4.6	792
Google Slides	83.74	10k-50k	Productivity	4.1	117,809
Google Calendar	33.25	100k-500k	Productivity	4.1	484,095

The results obtained by testing the applications in Table 1 are reported in Tables 2 and 3. In order to analyze whether the existing patterns were correctly implemented or not, the percentage of executed events and the amount of time the iMPAcT tool took during the exploration were gathered.

Table 2 indicates the results obtained when testing different patterns for each application:

- 'NA' - Not Applicable means that the Test Pattern was not applied. For instance, when the AUT does not have a side drawer the tool will not execute the corresponding Test Pattern;
- 'FF' - Failure Found means that at least one failure was found in the UI Pattern being tested;
- 'AF' - Absence of Failures means that the UI Pattern was detected but no failures were found.

The input value for the resource dependency pattern used in this case study was *Wifi*.

Table 2. Summary of test results part 1: FF-Failure Found; AF-Absence of Failures; NA-Not Applicable

Applications	SideDrawer	Orientation	Resource (wifi)	Tabs-Scroll
Book Catalog	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

It is known that the orientation change may reveal several failures [7]. In this experiment, the orientation test pattern was the one that detected more failures.

Some of the orientation failures found are:

- Book Catalog: if we rotate the screen, after a pop-up appears, such pop-up disappears (illustrated in Figure 6);
- Tippy Tipper: after pressing the *Calculate* button (Screen 1 on the FSM of Figure 5), the *Round Down* button disappears. In fact, it is still present but its text has changed to *Down* and, thus, it is not identified as being the same button;
- TomDroid: when the "More Options" menu is open, rotating the screen makes one of the options ("Search") disappear from the menu. The search button is now on top of the screen.

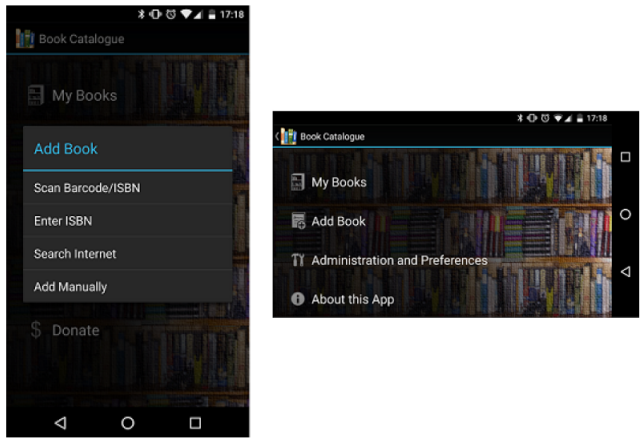

Fig. 6. Rotating the screen makes the dialog disappear.

This is depicted in Figure 7 and may not be a real error (the tester should decide if it is a development option or a problem);

- Calendar: when selecting the month, a calendar appears but rotating the screen makes it disappear; when opening the *More Options* menu, rotating the screen makes it disappear; and rotating the screen at any moment makes the action bar disappear;
- Google Slides: rotating the screen after opening the "More options" menu, makes the menu disappear.

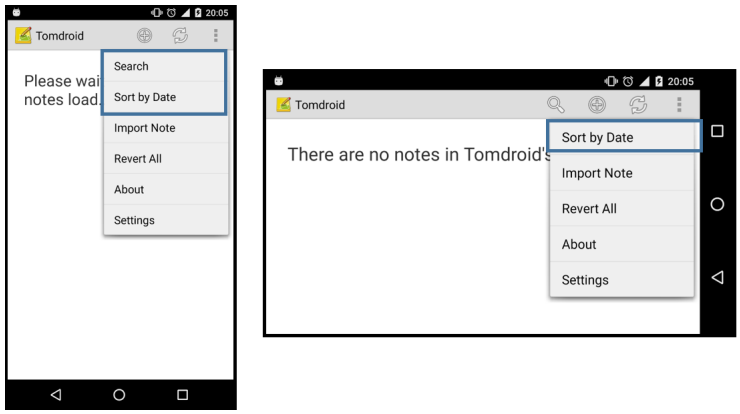

Fig. 7. Rotating the screen makes the Search option disappear

Considering the "tab scroll" pattern, examples of failures found are:

- Book Catalog: 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 8.

Considering the Side Drawer pattern, the failures found are:

- Calendar: the Side Drawer does not take up the full height of the screen. This is depicted in Figure 9.

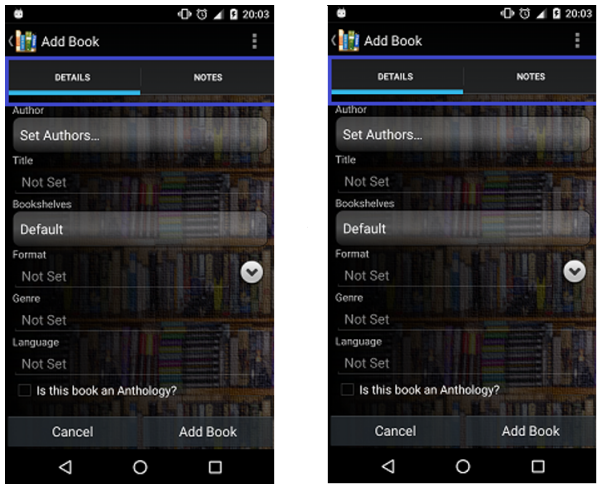

Fig. 8. Screen of Book Catalog that contains the tabs: “Details" and “Notes". Horizontally swiping the screen does not change the selected tab.

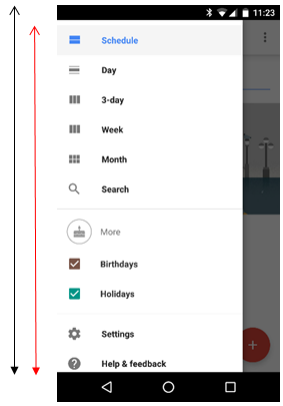

Fig. 9. The Side Drawer of the Calendar application does not take up the full height of the screen

The exploration stops when it reaches the home screen of the device, after pressing the *back button*. However, this does not guarantee the execution of all the events identified during the exploration. Moreover, since the process of selecting which events to fire has some randomness, different executions of the iMPAcT tool may traverse different paths of the app and may result in a different set of identified events. As such, each application was explored several times in order to obtain a more accurate percentage of executed events. Table 3 provides the average results of the several explorations:

- exploration time: average of the time taken by all explorations performed on each mobile application. This includes both the exploration and the testing (and test case generation) processes as they are intertwined;
- % of events: average of events fired over events identified in each exploration of each mobile application;

- % of events over all explorations: considering all the explorations performed on each application, calculates the average of events triggered over the maximum number of events identified.

The metrics presented in Table 3 provide some information on the coverage achieved by iMPAcT. Overall, it was possible to cover more than 46% of the events.

Table 3. Summary of test results part 2

Applications	% of events	% of events over all explorations	Exploration Time
Book Catalog	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

7 DISCUSSION

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. Even though the exploration may take some time when dealing with large applications (Google Slides took, in average, almost twenty minutes), the process is fully automatic and does not require any supervision.

The case study showed that the iMPAcT tool is capable of successfully identifying and testing the UI Patterns present in mobile applications. However, none of the applications tested presented failures in the *Resource Dependency* pattern. This enables us to respond affirmatively to **RQ1**.

Considering the percentage of events executed over the events identified in each exploration, all the values are over 60%. For the Tippy Tipper such percentage was even 93%. Considering the percentage of events fired over the maximum number of events identified on every explorations for a specific mobile application, such percentage is always over 45% (Table 3). With this data it is possible respond to the first part of **RQ2**: the coverage within the exploration is good but when considering the full amount of events identified throughout all of the explorations, we can conclude that iMPAcT's exploration algorithm can still be improved.

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 running iMPAcT, the answer to the second part of **RQ2** is affirmative. Nevertheless, the tester may stop the exploration whenever he wants by pressing manually the home button obtaining the results produced so far.

The main threat to validity of the approach is its dependence on the exploration algorithm and the intercalate of the exploration with the testing activity. For instance, it may be possible that a fired event removes an item from a list and, because of that, it is no longer possible to interact with the item, while in another execution trace the interaction with the item may take place before the item is removed. The same may happen after running the test strategies which may make it impossible to return to previous state. The goal of this paper is not to analyze different algorithms. However, [27] presents a study on how iMPAcT reacts to different exploration algorithms.

The iMPAcT tool combines MBT with automatic exploration (also known as crawlers or monkey testing). One of the challenges of MBT is the effort needed to build the model. This is the case, for

instance, of Costa *et al.*'s work [30], which, alike ours, tests UI Patterns on Android applications. However, they have to manually build the model of the AUT using their DSL described in [52], while our approach does not need any model built by the user. Moreover, the UI Patterns they are able to test can be found in several non-mobile applications while our patterns catalog is composed of UI patterns specific for the mobile world.

The main problem of crawlers is that they only identify crashes due to the lack of an oracle. One example of such a tool is the Ripper developed by Amalfitano *et al.*[4]. There are a few differences between the Ripper and the iMPAcT tool:

- 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;
- 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 traces, 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.

8 CONCLUSIONS

Considering the fast increase of the number of mobile applications available at Google's Play Store and of the importance these applications have in our daily lives, it is of high importance to ensure their correct functioning. As most companies and developers prefer to spend as little time as possible on the testing process, it is necessary to improve its automation.

The iMPAcT tool presented in this paper provides an automatic testing process that can be applied to any Android application without manually building an oracle. In order to do this, the iMPAcT tool automatically explores the application while trying to identify recurring behavior, *i.e.*, UI Patterns. These patterns are defined in a catalog and are associated to a set of Test Patterns, which indicate how the UI Pattern should be tested once it is detected. At the end, the tool produces a report containing the execution traces and the list of UI Patterns that lead to failures. It also builds a FSM that represents the exploration of the application in which it is possible to identify the paths that led to failures in order to ease their reproduction.

The main advantages of this approach are: 1) it does not require any manual effort being a full automatic process; 2) it does not require access to the source code of the application; 3) the catalog of patterns can be applied to any Android application and can be extended/improved by adding new patterns.

As future work we intend to further improve the exploration algorithm to reduce its randomness and to further improve the results' report. Also, at this point in time, and considering the gestures mentioned in [9], we are just simulating the onTouchEvent (with the click and the check) and the onLongPress (with long click). We aim to extend the user gestures' coverage in the future. Finally, the construction of the catalog of patterns is always an ongoing work.

REFERENCES

[1] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. 2015. Systematic execution of Android test suites in adverse conditions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis - ISSTA 2015*. ACM Press, New York, New York, USA, 83–93. https://doi.org/10.1145/2771783.2771786

[9] //developer.android.com/training/gestures/detector

[2] Pekka Aho, Tomi Raty, and Nadja Menz. 2013. Dynamic reverse engineering of GUI models for testing. In *2013 International Conference on Control, Decision and Information Technologies (CoDIT)*. IEEE, 441–447. https://doi.org/10.1109/CoDIT.2013.6689585

[3] Christopher W. Alexander, Sara Ishikawa, Murray Silverstein, and Max Jacobson. 1977. *A Pattern Language: Towns, Buildings, Construction* (1 ed.). Oxford University Press, New York, New York, USA. 1171 pages.

[4] Domenico Amalfitano, Nicola Amatucci, Anna Rita Fasolino, Ugo Gentile, Gianluca Mele, Roberto Nardone, Valeria Vittorini, and Stefano Marrone. 2014. 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)*. ACM, Västerás, Sweden, 29–34. https://doi.org/10.1145/2647648.2656426

[5] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. 2009. Experimenting a reverse engineering technique for modelling the behaviour of rich internet applications. In *2009 IEEE International Conference on Software Maintenance*. IEEE, 571–574. https://doi.org/10.1109/ICSM.2009.5306391

[6] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. 2012. Using GUI ripping for automated testing of Android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*. ACM Press, New York, New York, USA, 258. https://doi.org/10.1145/2351676.2351717

[7] Domenico Amalfitano, Vincenzo Riccio, Ana C. R. Paiva, and Anna Rita Fasolino. 2018. Why does the orientation change mess up my Android application? From GUI failures to code faults. *Softw. Test., Verif. Reliab.* 28, 1 (2018). https://doi.org/10.1002/stvr.1654

[8] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE '12*. ACM Press, New York, New York, USA, 1. https://doi.org/10.1145/2393596.2393666

[9] Google Android. 2008. Announcing the Android 1.0 SDK, release 1. http://goo.gl/5PSQHj

[10] Google Android. 2015. Android - What To Test. http://goo.gl/AL22tJ

[11] Google Android. 2015. Android Navigation Drawer. http://goo.gl/nnJOoj

[12] Google Android. 2015. Up and running with material design. https://goo.gl/GmsJSJ

[13] Google Android. 2016. Pure Android. http://goo.gl/LqNPyS

[14] Google Android. 2016. Tabs. https://www.google.com/design/spec/components/tabs.html

[15] Apple. 2007. Apple Reinvents the Phone with iPhone. https://goo.gl/AoFdyx

[16] Stephan Arlt, Cristiano Bertolini, and Martin Schäf. 2011. 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)*. Washington, DC, USA, 222–231.

[17] Andrea Avancini and Mariano Ceccato. 2013. Security testing of the communication among Android applications. In *8th International Workshop on Automation of Software Test (AST 2013)*. IEEE Press, 57–63. http://dl.acm.org/citation.cfm?id=2662413.2662427

[18] Leonid Batyuk, Markus Herpich, Seyit Ahmet Camtepe, Karsten Raddatz, Aubrey-Derrick Schmidt, and Sahin Albayrak. 2011. 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*. IEEE, 66–72. https://doi.org/10.1109/MALWARE.2011.6112328

[19] Nick Butcher and Android Developers. 2016. Android Design in Action: Navigation Anti-Patterns. https://www.youtube.com/watch?v=Sww4omntVjs

[20] Nick Butcher and Roman Nurik. 2016. Android Design in Action - Navigation anti - Patterns. http://www.allreadable.com/ff647Z8N

[21] Capgemini, HP, and Sogeti. 2015. *World Quality Report 2015-16*. Technical Report. https://goo.gl/SVoKtl

[22] Capgemini, Hp, Sogeti, and Hp. 2014. *World Quality Report 2014-15*. Technical Report. 1–64 pages. http://goo.gl/jzN2aA

[23] E.J. Chikofsky and J.H. Cross. 1990. Reverse Engineering and Design Recovery: a Taxonomy. *IEEE Software* 7, 1 (1990), 13–17. https://doi.org/10.1109/52.43044

[24] Inês Coimbra Morgado and Ana C. R. Paiva. 2015. Test Patterns for Android Mobile Applications. In *20th European Conference on Pattern Languages of Programs (Europlop 2015)*. Irsee, Germany. http://dl.acm.org/citation.cfm?id=2855354

[25] Inês Coimbra Morgado and Ana C. R. Paiva. 2015. Testing approach for mobile applications through reverse engineering of UI patterns. In *Sixth International Workshop on Testing Techniques for Event BasED Software*.

[26] Inês Coimbra Morgado and Ana C. R. Paiva. 2015. The iMPAcT Tool: Testing UI Patterns on Mobile Applications. In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015)*. Lincoln, NE, USA. http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=7372083

[27] Inês Coimbra Morgado and Ana C. R. Paiva. 2016. Impact of execution modes on finding Android failures. *The 7th International Conference on Ambient Systems, Networks and Technologies* 83 (2016), 284–291. https://doi.org/10.1016/j.procs.2016.04.127

[28] Inês Coimbra Morgado, Ana C. R. Paiva, and João Pascoal Faria. 2011. Reverse Engineering of Graphical User Interfaces. In *The Sixth International Conference on Software Engineering Advances (ICSEA '11)*. Barcelona, 293–298.

[29] Inês Coimbra Morgado, Ana C. R. Paiva, and João Pascoal Faria. 2012. Dynamic Reverse Engineering of Graphical User Interfaces. *International Journal On Advances in Software* 5, 3 and 4 (2012), 224–236. http://goo.gl/yRoIKF

[30] Pedro Costa, Ana C. R. Paiva, and Miguel Nabuco. 2014. Pattern Based GUI Testing for Mobile Applications. In *9th International Conference on the Quality of Information and Communications Technology (QUATIC 2014)*. IEEE, Guimarães, Portugal, 66–74. https://doi.org/10.1109/QUATIC.2014.16

[31] Marco Cunha, Ana C R Paiva, Hugo Sereno Ferreira, and Rui Abreu. 2010. PETTool: A pattern-based GUI testing tool. In *Software Technology and Engineering (ICSTE), 2010 2nd International Conference on*, Vol. 1. IEEE, San Juan, PR, V1–202 – VI–206. https://doi.org/10.1109/ICSTE.2010.5608882

[32] Muneer Ahmad Dar and Javed Parvez. 2014. Enhancing security of Android & IOS by implementing need-based security (NBS). In *2014 International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT)*. IEEE, 728–733. https://doi.org/10.1109/ICCICCT.2014.6993055

[33] Apple Developer. 2016. iOS Human Interface Guidilines. https://goo.gl/kUhwJE

[34] Dominik Franke, Corinna Elsemann, Stefan Kowalewski, and Carsten Weise. 2011. Reverse Engineering of Mobile Application Lifecycles. In *18th Working Conference on Reverse Engineering (WCRE '11)*. IEEE, 283–292. https://doi.org/10.1109/WCRE.2011.42

[35] Dominik Franke, Stefan Kowalewski, Carsten Weise, and Nath Prakobkosol. 2012. Testing Conformance of Life Cycle Dependent Properties of Mobile Applications. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 241–250. https://doi.org/10.1109/ICST.2012.104

[36] Yanick Fratantonio, Aravind Machiry, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. 2015. CLAPP: characterizing loops in Android applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*. ACM Press, New York, New York, USA, 687–697. https://doi.org/10.1145/2786805.2786873

[37] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. 2014. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*. ACM Press, New York, New York, USA, 1025–1035. https://doi.org/10.1145/2568225.2568276

[38] A. M. P. Grilo, A. C. R. Paiva, and J. P. Faria. 2010. Reverse engineering of GUI models for testing. In *The 5th Iberian Conference on Information Systems and Technologies (CISTI '10)*. IEEE, 1–6. http://goo.gl/bXcIy

[39] Daniel R. Hackner and Atif M. Memon. 2008. Test case generator for GUITAR. In *Companion of the 13th international conference on Software engineering (ICSE Companion '08) (ICSE Companion '08)*. ACM Press, New York, New York, USA, 959. https://doi.org/10.1145/1370175.1370207

[40] Konstantin Holl and Frank Elberzhager. 2014. A Mobile-Specific Failure Classification and Its Usage to Focus Quality Assurance. In *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE, 385–388. https://doi.org/10.1109/SEAA.2014.19

[41] Cuixiong Hu and Iulian Neamtiu. 2011. Automating GUI testing for android applications. In *6th International Workshop on Automation of Software Test (AST 2011)*. ACM, 77–83. https://doi.org/10.1145/1982595.1982612

[42] Gennaro Imparato. 2015. A combined technique of GUI ripping and input perturbation testing for Android apps. In *37th International Conference on Software Engineering - Volume 2 (ICSE '15)*. IEEE Press, 760–762. http://dl.acm.org/citation.cfm?id=2819009.2819159

[43] Nathan Ingraham. 2013. Apple announces 1 million apps in the App Store, more than 1 billion songs played on iTunes radio. http://goo.gl/z3RprB

[44] ISO/IEC. 2011. *ISO/IEC 25010:2011 - Systems and software engineering âĂŤ Systems and software Quality Requirements and Evaluation (SQuaRE) âĂŤ System and software quality models*. Technical Report. https://www.iso.org/obp/ui/{#}iso:std:iso-iec:25010

[45] Casper S. Jensen, Mukul R. Prasad, and Anders Møller. 2013. Automated testing with targeted event sequence generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis - ISSTA 2013*. ACM Press, New York, New York, USA, 67. https://doi.org/10.1145/2483760.2483777

[46] Mona Erfani Joorabchi and Ali Mesbah. 2012. Reverse Engineering iOS Mobile Applications. In *2012 19th Working Conference on Reverse Engineering*. IEEE, 177–186. https://doi.org/10.1109/WCRE.2012.27

[47] Antti Kervinen, Mika Maunumaa, Tuula Pääkkönen, Mika Katara, Wolfgang Grieskamp, and Carsten Weise. 2005. Model-Based Testing Through a GUI. In *5th International Workshop on Formal Approaches to Testing of Software (FATES 2005) (Lecture Notes in Computer Science)*, Wolfgang Grieskamp and Carsten Weise (Eds.), Vol. 3997. Springer-Verlag Berlin, Berlin, Heidelberg, 16–31. https://doi.org/10.1007/11759744

[48] Yuta Maezawa, Hironori Washizaki, and Shinichi Honiden. 2012. Extracting Interaction-Based Stateful Behavior in Rich Internet Applications. In *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE, 423–428. https://doi.org/10.1109/CSMR.2012.53

[49] Riyadh Mahmood, Naeem Esfahani, Thabet Kacem, Nariman Mirzaei, Sam Malek, and Angelos Stavrou. 2012. A whitebox approach for automated security testing of Android applications on the cloud. In *7th International Workshop on Automation of Software Test (AST 2012)*. IEEE, 22–28. https://doi.org/10.1109/IWAST.2012.6228986

[50] A Marchetto, P Tonella, and F Ricca. 2010. Under and Over Approximation of State Models Recovered for Ajax Applications. In *2010 14th European Conference on Software Maintenance and Reengineering*. IEEE, 236–239. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5714441

[51] Tiago Monteiro and Ana C. R. Paiva. 2013. Pattern Based GUI Testing Modeling Environment. In *Sixth IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2013)*. IEEE, Luxembourg, Luxembourg, 140–143. https://doi.org/10.1109/ICSTW.2013.24

[52] Rodrigo M. L. M. Moreira and Ana C. R. Paiva. 2014. 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.

[53] Rodrigo M. L. M. Moreira and Ana C. R. Paiva. 2014. 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)*. ACM Press, New York, New York, USA, 863–866. https://doi.org/10.1145/2642937.2648618

[54] Rodrigo M. L. M. Moreira, Ana C. R. Paiva, and Atif Memon. 2013. A pattern-based approach for GUI modeling and testing. In *24th IEEE International Symposium on Software Reliability Engineering (ISSRE 2013)*. IEEE, Passadena, CA, 288–297. https://doi.org/10.1109/ISSRE.2013.6698881

[55] Rodrigo M. L. M. Moreira, Ana C. R. Paiva, Miguel Nabuco, and Atif Memon. 2017. Pattern-based GUI testing: Bridging the gap between design and quality assurance. *Softw. Test., Verif. Reliab.* 27, 3 (2017). https://doi.org/10.1002/stvr.1629

[56] Henry Muccini, Antonio di Francesco, and Patrizio Esposito. 2012. Software testing of mobile applications: Challenges and future research directions. In *7th International Workshop on Automation of Software Test (AST 2012)*. IEEE, Zurich, Switzerland, 29–35. https://doi.org/10.1109/IWAST.2012.6228987

[57] Miguel Nabuco, Ana C.R. Paiva, Rui Camacho, and João Pascoal Faria. 2013. Inferring UI patterns with Inductive Logic Programming. In *8th Iberian Conference on Information Systems and Technologies (CISTI '13)*. Lisbon, Portugal, 1–5.

[58] Miguel Nabuco and Ana C. R. Paiva. 2014. Model-Based Test Case Generation for Web Applications. In *14th International Conference on Computational Science and Applications (ICCSA 2014)*.

[59] Theresa Neil. 2014. *Mobile Design Pattern Gallery: UI Patterns for Smartphone Apps* (2nd ed.). O'Reilly Media, Inc., Sebastopol, Canada.

[60] Cu D. Nguyen, Alessandro Marchetto, and Paolo Tonella. 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*. ACM Press, New York, New York, USA, 100. https://doi.org/10.1145/2338965.2336765

[61] Erik G. Nilsson. 2009. Design patterns for user interface for mobile applications. *Advances in Engineering Software* 40, 12 (dec 2009), 1318–1328. https://doi.org/10.1016/j.advengsoft.2009.01.017

[62] Ana C. R. Paiva, João C. P. Faria, and Pedro M. C. Mendes. 2007. Reverse engineered formal models for GUI testing. In *The 12th international conference on Formal methods for industrial critical systems*. Springer-Verlag, 218–233. https://doi.org/10.1007/978-3-540-79707-4_16

[63] Tristan Ravitch, E. Rogan Creswick, Aaron Tomb, Adam Foltzer, Trevor Elliott, and Ledah Casburn. 2014. Multi-App Security Analysis with FUSE. In *Proceedings of the 4th Program Protection and Reverse Engineering Workshop on 4th Program Protection and Reverse Engineering Workshop - PPREW-4*. ACM Press, New York, New York, USA, 1–10. https://doi.org/10.1145/2689702.2689705

[64] A. Rohatgi, A. Hamou-Lhadj, and J. Rilling. 2008. An Approach for Mapping Features to Code Based on Static and Dynamic Analysis. In *2008 16th IEEE International Conference on Program Comprehension*. IEEE, 236–241. https://doi.org/10.1109/ICPC.2008.35

[65] Clara Sacramento and Ana C. R. Paiva. 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)*. IEEE, Guimarães, Portugal, 105–115. https://doi.org/10.1109/QUATIC.2014.20

[66] Alireza Sahami Shirazi, Niels Henze, Albrecht Schmidt, Robin Goldberg, Benjamin Schmidt, and Hansjörg Schmauder. 2013. 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*. ACM, 275–284. https://doi.org/10.1145/2480296.2480308

[67] Hossain Shahriar, Sarah North, and Edward Mawangi. 2014. Testing of Memory Leak in Android Applications. In *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*. IEEE, 176–183. https://doi.org/10.1109/HASE.2014.32

[68] Mark Utting and Bruno Legeard. 2006. *Practical Model-Based Testing: A Tools Approach* (1 ed.). Morgan Kaufmann Publishers, San Francisco, CA, USA.

[69] Heila van der Merwe, Brink van der Merwe, and Willem Visse. 2012. Verifying android applications using Java PathFinder. *ACM SIGSOFT Software Engineering Notes* 37, 6 (nov 2012), 1. https://doi.org/10.1145/2382756.2382797

[70] Mark Wilson. 2008. T-Mobile G1: Full Details of the HTC Dream Android Phone. http://goo.gl/6vqI4E

[71] Qing Xie. 2006. Developing cost-effective model-based techniques for GUI testing. In *28th international conference on Software engineering - (ICSE 2006)*. ACM Press, New York, New York, USA, 997–1000. https://doi.org/10.1145/1134285.1134473

[72] Wei Yang, Mukul R. Prasad, and Tao Xie. 2013. A Grey-Box Approach for Automated GUI-Model Generation of Mobile Applications. In *16th International Conference on Fundamental Approaches to Software Engineering (FASE'13)*. Rome, Italy, 250–265. https://doi.org/10.1007/978-3-642-37057-1_19

[73] Siena Yu and Shingo Takada. 2015. External Event-Based Test Cases for Mobile Application. In *Proceedings of the Eighth International C* Conference on Computer Science & Software Engineering - C3S2E '15*. ACM Press, New York, New York, USA, 148–149. https://doi.org/10.1145/2790798.2790822

Received October 2018; revised December 2018; accepted February 2019