Pattern Based GUI testing for Mobile Applications

Pedro Costa
Department of Informatics Engineering, Faculty of Engineering of the University of Porto
Porto, Portugal
ei10011@fe.up.pt

Miguel Nabuco
Department of Informatics Engineering, Faculty of Engineering of the University of Porto,
Porto, Portugal
miguelnabuco@fe.up.pt

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

***Abstract*—This paper presents a study aiming to assess the feasibility of using the Pattern Based GUI Testing approach, PBGT, to test mobile applications. PBGT is a new model based testing approach that aims to increase systematization, reusability and diminish the effort in modelling and testing. It is based on the concept of User Interface Test Patterns (UITP) that contain generic test strategies for testing common recurrent behaviour, the so-called UI Patterns, on GUIs through its possible different implementations after a configuration step. Although PBGT was developed having web applications in mind, it is possible to develop drivers for other platforms in order to test a wide set of applications. However, web and mobile applications are different and only the development of a new driver to execute test cases over mobile applications may not be enough. This paper describes a study aiming to identify the adaptations and updates the PBGT should undergo in order to test mobile applications.**

I. Introduction

Mobile applications are seeing an incredible growth. According to Gartner Inc.[2], in 2013, 879.8 million mobile devices with Android operating system were shipped, and in 2014 this number is expected to grow to over one billion. Every month, around 20,000 new applications are released, and the current number of apps in the Android market is over 1,200,000 [3]. These numbers prove that this platform has reached a great sucess and that the developers are focusing more and more on developing applications for mobile operating systems.

However, due to particularities of the Android environment, the Android application testing process is a challenging activity [5]. Physical constraints of mobile devices (such as small memory, small display or low-power CPU) as well as developers lack of experience with the Android environment (which revolves around concepts such as activities, services and content providers, who do not exist in desktop or web applications) make mobile applications prone to new kinds of bugs [6]. A study found that Android applications have defect densities orders of magnitude higher than the operating system [7].

In this paper, we apply a methodology, Pattern-Based GUI Testing (PBGT) [1], that intends to test mobile applications based on models built upon User Interface Test Patterns (UITPs) [8]. This methodology was originally applied to test web applications.

Users interact with mobile applications in a different way they interact with web applications. In web applications users typically use a keyboard and a mouse whereas in mobile applications users only use their fingers. The range of possible actions is also diferent; in web applications users can only click, double-click, right-click or scroll, whether in mobile applications the range of possible actions is broader and includes long-press, pinch to zoom, and swipe.

This paper aims to identify the main differences between web and mobile applications in order to check if the User Interface Test Patterns (UITP) used for testing web applications still apply to testing mobile applications. It also attempts to answer if there is the need for aditional UITPs or, on the contrary, if some existing UITPs do not apply to mobile applications.

Finally, the effectiveness of the PBGT approach for finding errors on mobile applications will be assessed on a mutated open-source application.

This paper is structured as follows: in Section II, the state of the art regarding model-based testing and mobile testing is presented. Section III explains the overall architecture of PBGT. Section IV presents the case study: the research questions, the application under test, the mutants and the discussion of the results. Finally, Section V presents the conclusions reached, as well as future work.

II. State of the art

Model-based testing is the automation of the design of black-box tests [4]. It allows the creation of a model with the expected SUT behavior, rather than manually writing the tests, and then deriving test cases from that same model.

Although discussed and researched actively for some years, MBT has not been widely used in the industry. Major obstacles in the adoption of MBT include organizational difficulties and the lack of easy-to-use tools [17] [18]. However, MBT has shown to be effective in detection of errors [13][11][12].

There are multiple MBT approaches such as Spec#, VAN4GUIM, Event Flow Graphs (EFGs), Labeled State Transition Systems (LSTSs) and Finite State Machines (FSMs).

A GUI mapping tool was developed in [9], where the GUI model was written in Spec# with state variables to model the state of the GUI and methods to model the user actions on the GUI. However, the effort required for the construction of Spec# GUI models was too high. An attempt to reduce the time spent with GUI model construction was described in [10] where a visual notation (VAN4GUIM) is designed and translated to Spec# automatically. The aim was to have a visual front-end that could hide Spec# formalism details from the testers.

Event Flow Graphs is one of the most popular approach. The idea is to use a Ripping Tool [15] in order to automatically create a model that features all possible event interactions.

Other approach uses Labelled State Transition Systems (LSTSs), action words and keywords, with the goal to describe a test model as a Labelled Transition System (LTS), where its transactions correspond to action words [19]. Keywords correspond to key presses and menu navigation. According to the authors, by varying the order of events it becomes possible to find previously undetected events.

Miao et al. [22] proposed a FSM approach (called GUI Test Automation Model – GuiTam) with the goal to overcome some limitations of the EFG approach. They proved that there exists an inclusive mapping between EFG and GuiTam, that is, for each EFG, there exists at least one GuiTam that is able to automate all the EFG automated tests. They also proved that the storage requirement of the GuiTam is one order less than that of the EFG model, and the computational complexity is at a similar level.

Concerning model-based testing on mobile applications, more specifically on Android applications, there are not many model-based testing approaches, as this is a relatively recent technology. Yumei et al. [14] proposed an Optical Character Recognition (OCR) ripper, that uses OCR to traverse through the application GUI in order to create a GUI model and generate test cases. This process is not fully automatic, as the OCR ripper sometimes provides innacurate results, or misses GUI windows. Their aproach is OS independent, so it can be used in any mobile operating system.

Takala [16] developed TEMA Tools, which is a set of model-based testing tools, that includes tools for test modeling, test design, test generation and test debugging. TEMA models are state machines, specifically labeled state transition machines. Although TEMA Tools was developed for testing Symbian GUI applications, it was adapted to Android by adding a keyword-based test automation tool, that is able to detect GUI elements through keywords stored in Android Window service.

Wang [20] presents a grey-box approach for automatically extracting a model of a given mobile application. In his approach, he extracts the set of events supported by the application GUI, using static analysis. Then, by systematically running these events on the running application using a crawler, he constructs a model of the application. Although this approach is almost fully automatic, it requires a good knowledge of the precise set of GUI actions to generate a high quality model.

Most of these approaches are tied to the application to test. The application must be given to the tool in order to generate the model. This does not promote reusability. Other approaches do not generate test scripts forcing the tester to build them. Most of these test scripts have only one path and the models do not allow cycles resulting in bigger models. In addition, MBT approaches usually use models describing the expected behaviour of the application under test instead of describing the testing goals in a higher level of abstraction.

PBGT approach aims to solve these issues by providing UITP (User Interface Test Patterns) that are generic test strategies to test common recurrent behaviour on GUIs over their different possible implementations after a configuration phase. By using UITP, PBGT increases the level of abstraction of test models, promotes reuse and diminishes the effort required in the modelling phase of the model based testing process.

III. Overall architecture of PBGT

Pattern Based GUI Testing (PBGT) is a testing approach that aims to increase the level of abstraction of test models and to promote reuse [1] in order to diminish the model based testing effort. PBGT has the following main components:

- **PARADIGM-DSL** — A domain specific language (DSL) for building GUI test models based on UI test patterns (UITP) [25];
- **PARADIGM-ME** — A modelling and testing environment to support the building, mapping and configuration of test models [26];
- **PARADIGM-TG** — A test case generation tool that builds test cases from PARADIGM models according to different coverage criteria [27];
- **PARADIGM-TE** — A test case execution tool to execute the tests, analyze the coverage (on the model and on the code) (PARADIGM-COV) and create reports with test results [24] ;
- **PARADIGM-RE** — A reverse engineering tool that extracts part of the PARADIGM models from existent web applications [28].

The activity diagram of the PBGT Tool set is shown in Figure 1.

Currently, PBGT is used to test web applications. By using PARADIGM-ME, the user can build and configure the test model written in PARADIGM language with UI Test patterns. Each UI Test Pattern instance may have multiple configurations. There are two types of configurations: Valid and Invalid. For example, for a login UI test pattern this includes an invalid configuration that simulates an erratic behavior of the user (for example, inserting wrong username and password for a denied authentication) and a valid authentication (for example, providing valid username and password for an acceptable authentication); and for an input test pattern it may include simulating an erratic behaviour by inserting letters in a text box that only accepts numbers and the correct behaviour by inserting numbers. In addition, it is possible to provide

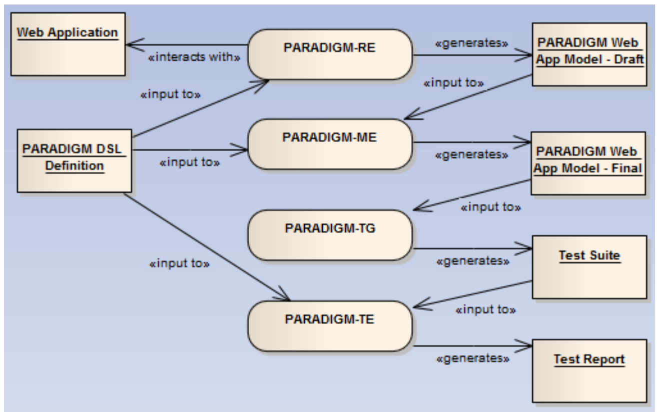


Fig. 1. Activity Diagram of the PBGT tool set

different configurations for testing valid (or invalid) behaviour as far as different test input data is provided.

A UI Test Pattern describes a generic test strategy, formally defined by a set of test goals, denoted as

$$< Goal; V; A; C; P > \tag{1}$$

where:

- **Goal** is the ID of the test;
- **V** is a set of pairs [variable, inputData] relating test input data (different for each configuration) with the variables involved in the test;
- **A** is the sequence of actions to be performed during test execution;
- **C** is the set of checks to be performed during test execution;
- **P** is the precondition defining the set of states in which is possible to perform the test.

The PARADIGM language (Figure 2) is comprised by elements and connectors [25] (Figure 3). There are four types of elements: Init (to mark the beginning of a model), End (to mark the termination of a model), Structural (to structure the models in different levels of abstraction), and Behavioral (UI Test Patterns describing the behavior to test).

As models become larger, coping with their growing complexity forces the use of structuring techniques such as different hierarchical levels that allow use one entire model "A" inside another model "B" abstracting the details of "A" when within "B". It is like what happens in programming languages, such as C and Java, with constructs such as modules. Form is a structural element that may be used for that purpose. A Form is a model (or sub-model) with an Init and an End elements.

Group is also a structural element but it does not have Init and End and, moreover, all elements inside the Group are executed in an arbitrary order. The PARADIGM's elements and connectors are described by: (i) an icon/figure to represent the element graphically and; (ii) a short label to name the element. The concrete syntax of the DSL is illustrated in Figure 3. Additionally, elements within a model have a number to identify them and, optional elements have a "op" label next to its icon/figure.

This language has three connectors (the definition of these connectors is based on ConcurTaskTrees - CTT [29]): "Sequence"; "SequenceWithDataPassing"; and "SequenceWithMovedData". The "Sequence" connector indicates that the testing strategy of the target element cannot start until the testing strategy of the source element has completed. The "SequenceWithDataPassing" connector has the same behavior as "Sequence" and, additionally, indicates that the target element receives data from the source element. "SequenceWithMovedData" has a similar meaning to the "SequenceWithDataPassing" connector, however, the source element transfers data to the target, so the source loses the data that was transferred. In addition, there is another kind of relation among elements – "Dependency" – indicating that the target element depends on the properties of a set of source elements, for instance, when it is the result of a calculation.

In PARADIGM-ME, the tester builds the web application test model, by creating the respective UITPs and joining them with connectors. These connectors define the order that the UITPs will be performed. PARADIGM models have to be written according to a set of rules in order to be consistent and allow generating test cases from it. These rules are implemented in the modelling environment as OCL constraints [25]. After building the model, the tester has to configure each UITP with the necessary data (test input data, pre-conditions, and checks). The list of UITPs supported by PARADIGM-DSL (Figure 2) is the following:

- **Login** — The Login UITP is used to verify user authentication. The goal is to check if it is possible to authenticate with a valid username/password and check if it is not possible to authenticate otherwise.
- **Find** — Find UITP is used to test if the result of a search is as expected (if it finds the right set of values).
- **Sort** — The Sort UITP is used to check if the result of a sort action is ordered accordingly to the chosen sort criterion (such as sort by name, by price and ascending or descending).
- **Master Detail** — Master Detail UITP is applied to elements with two related objects (master and detail) in order to verify if changing the master's value correctly updates the contents of the detail.
- **Input** — The Input UITP is used to test the behavior of input fields for valid and invalid input data.
- **Call** — The Call UITP is used to check the functionality of the corresponding invocation. It is usually a link that may lead to a different web page.

To execute the tests, it is necessary to establish a mapping between the model UITPs and the UI patterns present on the web application under test. This is performed by a point and click process starting by selecting the element within the UITP and then pointing the corresponding element in the GUI. This will allow the test execution module to know which web

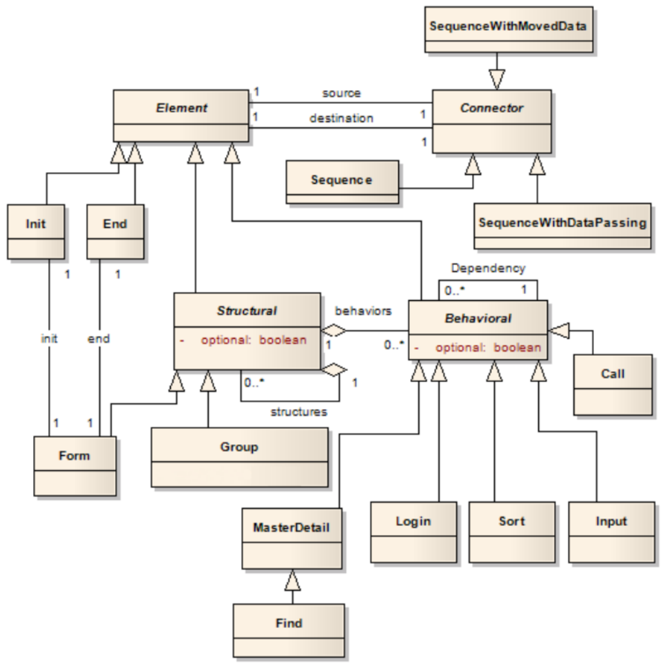


Fig. 2. Model of PARADIGM language

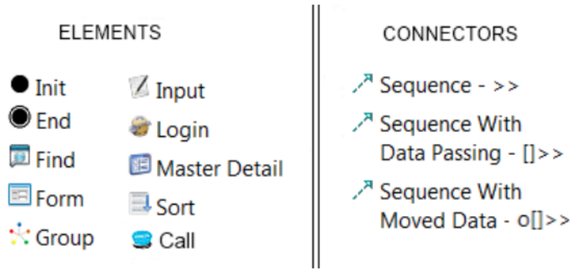


Fig. 3. PARADIGM syntax

elements to interact with that corresponds to a certain UITP described in the model. For the mapping, PARADIGM-ME will save the following information:

- **Text boxes ID's** — the ID property of the GUI object.
- **Images** — The image of the UITP form. This is saved through Sikuli [21] and is used when the tool is not able to identify the object by its ID.
- **Area coordinates** — The coordinates of the object. When the two previous methods fail, the tool uses the coordinates in order to interact with the object.

After the mapping is made, PARADIGM-TG will then generate test cases from the model. With the test cases generated, PARADIGM-TE will perform the tests on the web application and provide reports with the results.

The mapping process is performed manually by the tester which is prone to errors, either by selecting the wrong GUI element of selecting a set of elements instead of one. In this case the behaviour of the test may be different from expected. However, if this happen, the tester may fix the mapping information without changing the model and afterwards the process is automatic.

PBGT can be used to test web applications that are being developed from scratch or to test already existing web applications. In the latter scenario, PARADIGM-RE can be useful to generate part of the PARADIGM model by exploring the existing web application. Afterwards, the tester can validate and complete the model with additional configurations and/or additional (not found by the reverse engineering approach) UITPs [8].

The use of this tool can be seen on a video at www.fe.up.pt/~apaiva/tools/PARADIGM5min.mp4.

IV. Case study

Pattern Based GUI Testing approach was initially developed having in mind the testing of web applications and uses Selenium in its core.

In order to use PBGT to test mobile applications, a new driver was developed. This driver uses Selendroid, due to its high compatibility with Selenium, so most of the already existing PBGT code could be reused. The main differences between web and mobile versions of the PBGT approach were in the mapping and interaction strategy. The difference relies mainly on the attributes from each elements that are saved. For mobile applications, PBGT does not save the area coordinates of the element due to the large discrepancies in screen sizes and resolutions and to screen rotations. Nonetheless, relies more on the *id* of the element, if it has one, in order to interact with it during test case execution.

The aim of this case study is to assess the applicability of PBGT in testing mobile applications and, more specifically, indends to answer the following research questions.

A. Research questions

R1) Is it possible to use PBGT approach to test mobile applications?

R2) Is it possible to find bugs in mobile applications using the PBGT testing approach for testing mobile applications?

B. Application under test

The experiment was performed over one Android application: **Tomdroid**[23], a note-taking application. Tomdroid is an application that acts as a Tomboy client, which is a desktop note-taking application. Therefore, Tomdroid can synchronize its notes with Tomboy, allowing basic Create, Retrieve, Update and Delete (CRUD) operations and other functionalities like sorting notes and link to other notes, sites and phone numbers.

Tomdroid was chosen because it is an open-source application, allowing us to fault seeding mutants in the code and measure the effectiveness of the PBGT approach to test mobile applications.

C. Mutants

In order to assess the capacity of PBGT in finding failures on mobile applications we followed a fault seeding strategy in which we generated mutants that negated the Boolean conditions inside *if* and *while* clauses. Afterwards, we ran the test cases generated by PBGT in order to check if the mutants were killed or not.

Tomdroid is a Tomboy client for Android. Tomboy is an open-source desktop notetaking application. The Tomdroid packages chosen for fault seeding mutants were **org.tomdroid**, which is responsible for managing CRUD operations; and **org.tomdroid.ui**, the package responsible for user interaction.

PBGT was able to kill 85.4% of the mutants (see table VIII).

D. Threats to validity

This experiment was performed over a single mobile application. If we test the PBGT approach on a broader set of applications the mutation score could be different. In particular, it may happen when the application requires specific mobile interactions, like swipe, which we are not able to reproduce in PBGT at this moment.

E. Test Models

The test models were built using the PARADIGM-DSL language. As an example, Figure 4 shows one of the test models of the Tomdroid application. This model is used to test editing notes.

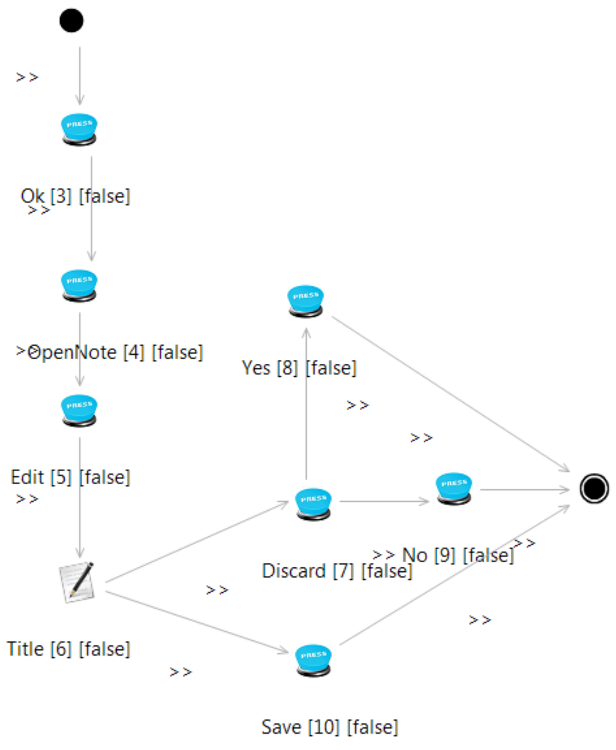


Fig. 4. Tomdroid model to test editing notes

This test model was built using only two different patterns: the Call UITP and the Input UITP. The sequence of actions performed by the Call UITP during test case execution is a click event. It is possible to provide different configurations in order to check the proper behaviour of the application after such click. The Input UITP provides the means to check the behaviour of the application after inserting valid and invalid input data on input fields.

The model is easy to understand although it has 3 different test paths. The tester only has to be concerned with the model since these test paths are generated by PARADIGM-TG. The test paths are as follows:

- 3, 4, 5, 6, 7, 8
- 3, 4, 5, 6, 7, 9
- 3, 4, 5, 6, 10

All test paths start by clicking on a button (UITP 3) to dismiss a message that is always shown when the Tomdroid application is opened for the first time, as can be seen in Figure 7. After that, one can open an existing note (UITP 4), which is created upon the installation of the application, attempt to edit the note (UITP 5) and change the title (UITP 6). The test paths diverge at this point. In one path the changes are saved (UITP 10) whereas, in the other two, the discard button is clicked (UITP 7). After clicking on the discard button, one path confirms (UITP 8), returning to the saved state, and the other one does not (UITP 9), continuing editing the note.

All different test paths run separately so the changes made on the GUI that are not saved (in a database or file) in one are not propagated to the others. This way, the order in which they run is not relevant.

All elements in the model are configured so as to define the test input data (when needed), the checks to perform and the precondition defining when to run the related test strategy. Table I and Table II show the configurations for all the elements in the model in Figure 4.

Taking the element [5] within Table I as an example, after clicking on the related button, PBGT checks if the text "Welcome to Tomdroid!" is present in the layout and checks if the text "<size:large>" is not present in the layout. Besides checking if text is or is not present on a layout, the Call UITP can also be configured to check if the click action resulted in changing the activity (called page in the web applications domain) or, on the contrary, if stayed in the same activity.

Taking the element [6] of Table II, after sending the text "Edited" to the related GUI object, no check is performed.

Two more models were created to test the mobile application: one model for the creation and visualization of notes, showed in Figure 5 and another for testing the sorting criteria of notes, showed in Figure 6. The configurations set for model in Figure 5 are shown in Table III and IV and the configurations set for model in Figure 6 are shown in table V, VI, VII.

The test paths for the model shown in Figure 5 are:

- 3, 4, 6, 7, 8, 9
- 3, 4, 6, 7, 8, 10

TABLE I
CONFIGURATIONS OF CALL ELEMENTS

UITP	Check	Pre-condition
[3]	PresentInPage("Tomdroid's First Note")	true
[4]	PresentInPage("Welcome to Tomdroid!")	true
[5]	PresentInPage("Welcome to Tomdroid!") and NotPresentInPage("<size:large>")	true
[7]	StayOnSamePage	true
[8]	NotPresentInPage("Edited")	true
[9]	PresentInPage("Edited")	true
[10]	ChangePage	true

TABLE II
CONFIGURATIONS OF INPUT ELEMENT

UITP	Input	Pre-condition
[6]	Edited	true

- 3, 4, 6, 7, 11, 12, 13, 8, 9
- 3, 4, 6, 7, 11, 12, 13, 8, 10

The model shown in Figure 6 only has one test path:

- 3, 4, 7, 8, 9, 10, 11

TABLE III
CONFIGURATIONS OF CALL ELEMENTS OF FIGURE 5

UITP	Check	Pre-condition
[3]	StayOnSamePage	true
[4]	ChangePage	true
[7]	PresentInPage("First Note")	true
[8]	PresentInPage("Welcome")	true
[9]	ChangePage	true
[10]	PresentInPage("Secret Text")	true
[11]	ChangePage	true
[13]	PresentInPage("First Note")	true

When the configuration for all models is complete, the tester has to establish the map between UITP and real UI patterns within the mobile application shown in Figure 7. After this, it is possibe to generate test cases and run them over the mobile application. If a test path has only UITP with one configuration each, there will be one test case executing each configuration defined. When a test path traverses a UITP with 2 (or n) configurations, there will be 2 (or n) test cases executing the different configurations.

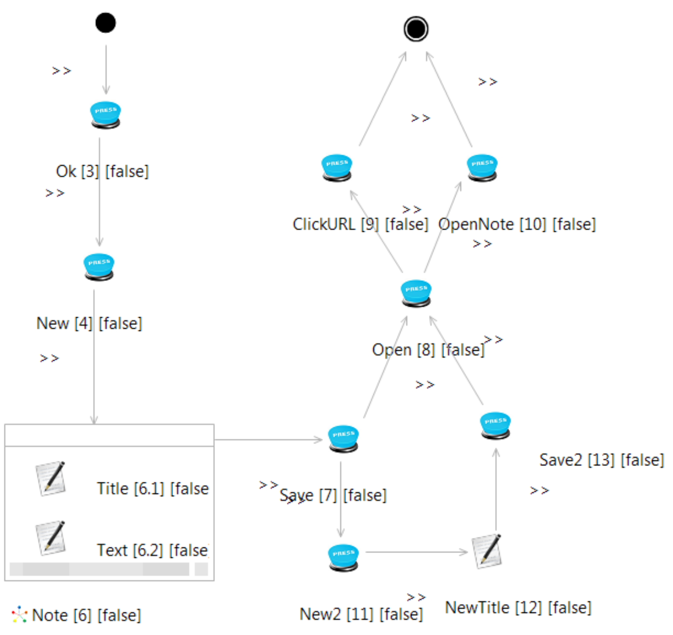


Fig. 5. Tomdroid model to test creating and viewing notes

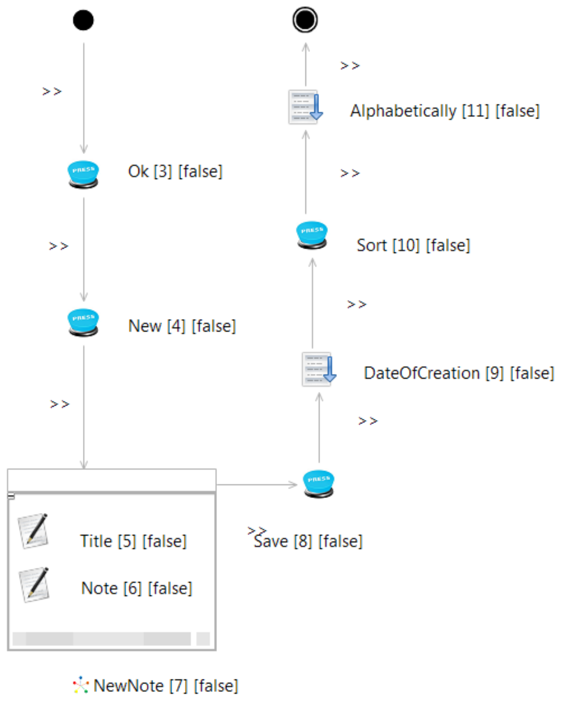


Fig. 6. Tomdroid model to test sorting notes

F. Failures Detected

Regarding failures detected, i.e., Tomdroid mutants killed, PBGT was able to detect a large number of failures, killing the majority of mutants as can be seen in table VIII. The mutation score is 85.4%.

TABLE IV
CONFIGURATIONS OF INPUT ELEMENTS OF FIGURE 5

UITP	Input	Pre-condition
[6.1]	Tomdroid	true
[6.2]	Secret Text	true
[12]	A	true

TABLE V
CONFIGURATIONS OF CALL ELEMENTS OF FIGURE 6

UITP	Check	Pre-condition
[3]	StayOnSamePage	true
[4]	ChangePage	true
[8]	ChangePage	true
[10]	StayOnSamePage	true

The failures detected can be classified into the following categories:

1) Notes do not open.
2) Application throws unhandled exceptions.
3) Notes do not show text.
4) Discarded changes are saved.
5) Notes are not saved.
6) Notes have unformatted content.
7) Notes are not created.
8) When creating new note with repeated title does not add a number.
9) When creating new note always adds a number.
10) Notes are displayed in the wrong order.
11) Malfunction in links between notes.
12) Wrong text in action bar.

These failures were detected by using the following checks: *a*) **PresentInPage**, which checks that some text is present in a layout; *b*) **NotPresentInPage**, which checks that some text is not present in a layout; *c*) **ChangePage**, which checks that after an action changed activity; or *d*) **StayOnSamePage**, which checks that after an action remained in the same activity.

Regarding category 1) (*Notes do not open*), it was possible to detect these failures by using the checks *a*) (*PresentInpage*) and *c*) (*ChangePage*). In some cases, by attempting to open a note, the activity does not change, failing in *c*); others, an error message is displayed, failing in *a*). In the latter case, *b*) could be used too with the same effect.

When an unhandled exception is thrown, the application stops, making it impossible to continue the model and failing the subsequent checks.

Failures within category 3) (*Notes do not show text*) are detected by the check *a*) (*PresentInPage*).

When editing a note, Tomdroid gives the user the possibility to save the changes or discard them. Failures such as 4) (*Discarded changes are saved*) and 5) (*Notes are not saved*) may arise with this type of behaviour. The model shown in Figure 4, with the configurations shown in Tables I and II, was able to detect this type of failure.

TABLE VI
CONFIGURATIONS OF INPUT ELEMENTS OF FIGURE 6

UITP	Input	Pre-condition
[5]	White Note	true
[6]	Secret Text	true

TABLE VII
CONFIGURATIONS OF SORT ELEMENTS OF FIGURE 6

UITP	Field	Pre-condition
[9]	[White Note, Tomdroid's First Note]	true
[11]	[Tomdroid's First Note, White Note]	true

Tomdroid's notes use an XML syntax in order to display the text in different ways. In item 6) (*Notes have unformatted content*) the text was displayed in that XML syntax and, with the check *b*) (*NotPresentInPage*) it was possible to detect this failure.

The user might create a note with a repeated title, thus making it hard to distinguish two different notes. When this happens, Tomdroid adds a number to the title making it different from the others. It was possible to detect failures such as Failures 7) (*Notes are not created*), 8) (*When creating new note with repeated title does not add a number*) and 9) (*When creating new note always adds a number*), by attempting to create the note and then using check *a*) (*PresentInPage*) to verify that the title is present and with the correct text.

Tomdroid allows the user to sort the notes displayed by title or by date of creation. For detecting the failure 10) (*Notes are displayed in wrong order*) it was used an element, the **Sort** element, that enabled to check the order in which the notes are displayed.

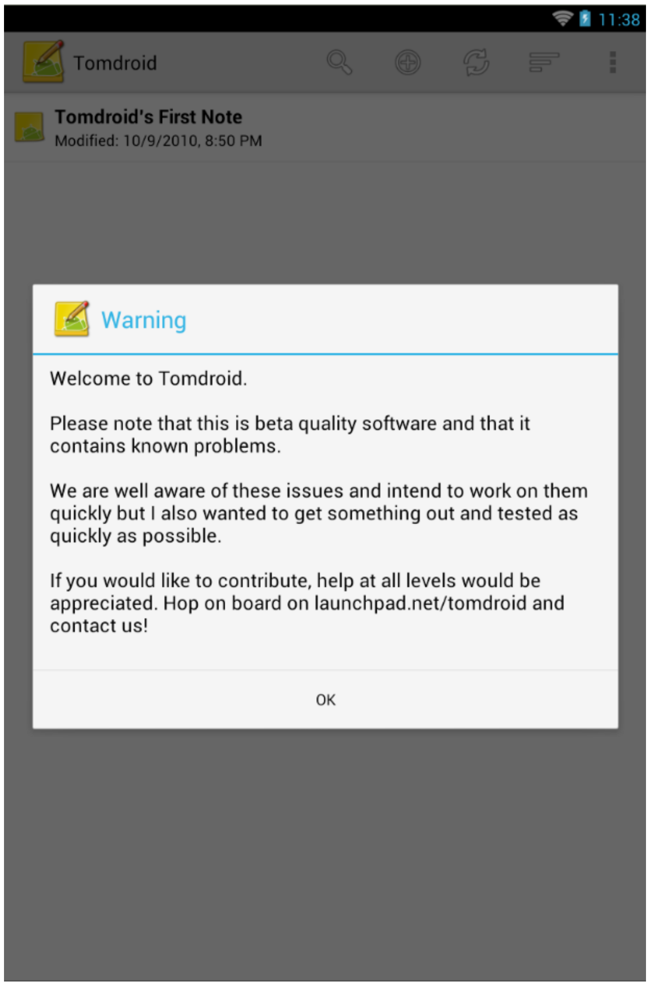
When a title of a note is written on the text of another note, a link is created. Then, the user can simply click on that link in order to open the note. With PBGT, it was possible to detect that a link was not working properly, Failures 11, by using check *c*) (*ChangePage*).

When the wrong text is shown in the action bar, Failures 12, they are detected using check *a*) (*PresentInPage*).

There were six mutants that PBGT was not able to kill. These mutants fit in two categories:

1) Notes do not link to another applications.
2) Text appears on the screen in a different color.

Regarding the first category, PBGT should be able to detect this failure with check *c*) nonetheless, PBGT was not able to detect these failures due to technical limitations, as Selendroid is only capable of interacting with one application and because



Welcome to Tomdroid.

Please note that this is beta quality software and that it contains known problems.

We are well aware of these issues and intend to work on them quickly but I also wanted to get something out and tested as quickly as possible.

If you would like to contribute, help at all levels would be appreciated. Hop on board on launchpad.net/tomdroid and contact us!

Fig. 7. Screenshot of Tomdroid application

TABLE VIII
TEST RESULTS USING PARADIGM

	Tomdroid
Mutants	41
Killed	35(85.4%)

of that not capable to detect if a new application was open.

Concerning the last category, PBGT does not have a mean to check text color.

G. Discussion

After the experiment we were able to answer the question R1) and R2). Indeed, besides PBGT was developed for the web context, it is possible to use PBGT for testing mobile applications and it is possible to find bugs on those applications. To use the PBGT approach to test mobile applications, some adjustments had to be made as described in section IV. The high percentage of mutants killed answers R2).

In addition to the above research questions, we were also able to gather some information regarding two other questions, not derived directly from the results of the experiment, but from the interaction with PBGT applyed to mobile applications. The questions are:

R3) Are the current UITPs enough for testing mobile applications?
R4) Are there specific ways of interaction particular to mobile applications that are not covered by PBGT?

Answering question R4), mobile applications give the user different ways of interaction, such as the longpress, the swipe and the pinch. These differences exist because users interact with mobile GUIs using their fingers rather than a mouse. Therefore, mobile applications do not have hover menus making the point sequence mapping mode of PBGT obsolete. There is the need for adding new sequence actions to the existing elements specific for mobile applications that include such different ways of interaction.

The different screen sizes between smartphones and tablets are a problem too. Some applications have different layouts because of this issue. Therefore, the model should behave according to these differences. One way of achieving this is by adding pre-conditions to the UITP elements specifying the screen sizes and resolutions they support. Then, the tester could specify the range of screen sizes he wants to test or simply let the tool choose a set of screen sizes that is sufficient to transverse all the model. These test strategies impose the use of an emulator as a real device has a fixed screen size.

Mobile applications give the user the possibility to rotate the screen, changing its orientation. Since on Android applications the Activity is destroyed and recreated on the event of changing the screen's orientation, several problems may arise. Some information may not be properly saved before the destruction of the Activity, making the recreated Activity outdated. Also, an uncaught exception may be thrown.

Lastly, since layouts may be different depending on the screen orientation and PBGT does not support it, it is impossible to test all layouts. This answers question R3), there is the need for a new UITP that changes the screen orientation and checks the state of the GUI.

Another difference between web and mobile applications is that it is common for mobile applications call other applications. It is common for an application to open a browser or the smartphone's camera for example.

As PBGT uses Sikuli, it is possible to interact with some elements that are not acessible using Selendroid alone. As an example, Tomdroid creates links inside TextViews that Selendroid can not interact with and, with Sikuli, PBGT can.

Nonetheless, PBGT does not have a mean to check text or another element's color.

It is also important to notice that this approach tests software applications through their GUIs which means that it does not test if the application data is effectivelly correct, i.e., it may happen that data seems correct through the GUI but it is not correct in the related database.

V. Conclusions and Future Work

This paper presented a case study that aimed to test the applicability of the PBGT approach to mobile applications. We started by fault seeding mutants into an open-source Android application named Tomdroid, in order to check how PBGT would behave in the mobile context. Then, we detailed the categories of failures that PBGT was able to detect and discussed the results.

We concluded that the PBGT approach works in the mobile context but still needs some improvements. As it was expected, mobile applications are different from web applications. The main differences are in the way the user interacts with the GUI, the possibility to change the screen's orientation and the possibility to have different layouts for different screen sizes.

In the future, PBGT needs to be able to emulate the different gestures the user typically can perform in a mobile application such as the swipe, also known as the flick; the longpress which happens when the user leaves the finger pressed for a longer period of time; and the pinch, the typical gesture for zooming in and out. These mobile gestures may be integrated in the Call UITP.

To solve the difference in screen sizes, as stated in the IV-G subsection, we will create new pre-conditions that may be added to the elements so the tester can specify on which screen sizes each element is present. This way, it is possible to test all different screen sizes with only one model.

Finally, we will create a new UITP for the mobile context that will allow to rotate the screen and provide it with some checks such as the *PresentInPage* and the *NotPresentInPage*.

It is also interesting for PBGT to support another Mobile OSs (such as iOS) which can be done by building specific drivers. The PBGT approach is all about reusability and the models could be used to test different applications in different OSs.

References

[1] Moreira, R. and Paiva, A. and Memon, A., “A Pattern-Based Approach for GUI Modeling and Testing”. Proceedings of the 24th annual International Symposium on Software Reliability Engineering (ISSRE 2013), 2013

[2] Gartner, "Gartner Says Worldwide Traditional PC, Tablet, Ultramobile and Mobile Phone Shipments Are On Pace to Grow 6.9 Percent in 2014", March, 2014, http://www.gartner.com/newsroom/id/269231, last acessed April 2014

[3] AppBrain, "AppBrain Stats", https://www.appbrain.com/stats/number-of-android-apps, last acessed April 2014

[4] Utting, Mark, and Bruno Legeard. Practical model-based testing: a tools approach. Morgan Kaufmann, 2010.

[5] Amalfitano, Domenico and Fasolino, Anna Rita and Tramontana, Porfirio and De Carmine, Salvatore 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, New York, NY, USA, 258-261.

[6] Hu, Cuixiong and Neamtiu, Iulian. 2011. Automating GUI testing for Android applications. In Proceedings of the 6th International Workshop on Automation of Software Test (AST '11). ACM, New York, NY, USA, 77-83.

[7] Maji, A. Kumar and Hao, K. and Sultana, S. and Bagchi, S. Characterizing failures in mobile oses: A case study with android and symbian. In Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on, pages 249-258.

[8] Nabuco, M.; Paiva, A.C.R.; Camacho, R.; Faria, J.P., "Inferring UI patterns with Inductive Logic Programming," Information Systems and Technologies (CISTI), 2013 8th Iberian Conference on, 19-22 June 2013

[9] Paiva, Ana C. R. and Faria, João C. P. and Tillmann, Nikolai and Vidal, Raul F. A. M., "A Model-to-Implementation Mapping Tool for Automated Model-Based GUI Testing", ICFEM, Lecture Notes in Computer Science, Springer, vol. 3785, pp. 450-464, 2005

[10] Moreira, Rodrigo M. L. M. and Paiva, Ana C. R., "Visual Abstract Notation for Gui Modelling and Testing - VAN4GUIM", ICSOFT 2008, pp. 104-111, 4 March 2008.

[11] Paiva, A.C.R., "Automated Specification-Based Testing of Graphical User Interfaces", Ph.D, Engineering Faculty of Porto University (Ph.D thesis), Department of Electrical and Computer Engineering (2007), www.fe.up.pt/ apaiva/PhD/PhDGUITesting.pdf

[12] A. C. R. Paiva, J. C. P. Faria, and R. F. A. M. Vidal, "Specification-based testing of user interfaces," in Interactive Systems. Design, Specification, and Verification, 10th International Workshop, June 2003, pp. 139-153.

[13] El-Far, Ibrahim K., and James A. Whittaker. "Model-Based Software Testing." Encyclopedia of Software Engineering (2001).

[14] Wu, Yumei, and Zhifang Liu. "A Model Based Testing Approach for Mobile Device." Industrial Control and Electronics Engineering (ICICEE), 2012 International Conference on. IEEE, 2012.

[15] Memon, Atif M., Ishan Banerjee, and Adithya Nagarajan. "GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing." WCRE. Vol. 3. 2003.

[16] Takala, T.; Katara, M.; Harty, J., "Experiences of System-Level Model-Based GUI Testing of an Android Application", Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on, pp.377,386, 21-25 March 2011

[17] Robinson, Harry. "Obstacles and opportunities for model-based testing in an industrial software environment." Proceedings of the 1st European Conference on Model-Driven Software Engineering, Nuremberg, Germany. 2003.

[18] A. Hartman, "AGEDIS project final report, 2004," Available at http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.91.575&rep=rep1&type=pdf, last acessed April 2014

[19] Kervinen, Antti, et al. "Model-based testing through a GUI." Formal Approaches to Software Testing. Springer Berlin Heidelberg, 2006. 16-31.

[20] Yang, Wei and Prasad, Mukul R. and Xie, Tao. 2013. A grey-box approach for automated GUI-model generation of mobile applications. In Proceedings of the 16th international conference on Fundamental Approaches to Software Engineering (FASE'13)

[21] Sikuli API, https://code.google.com/p/sikuli-api/, last acessed April 2014

[22] Miao, Yuan, and Xuebing Yang. "An FSM based GUI test automation model." Control Automation Robotics & Vision (ICARCV), 2010 11th International Conference on. IEEE, 2010.

[23] Tomdroid, https://launchpad.net/tomdroid, last acessed April 2014

[24] Liliana Vilela, Ana C. R. Paiva, "PARADIGM-COV - A Multimensional Test Coverage Analysis Tool", in CISTI 2014 - 9th Iberian Conference on Information Systems and Technologies, Barcelona, 18-21 June, 2014

[25] Rodrigo M. L. M. Moreira, Ana C. R. Paiva, "A GUI Modeling DSL for Pattern-Based GUI Testing - PARADIGM", 9th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2014), 28-30 April, Lisbon, Portugal

[26] Tiago Monteiro, Ana C. R. Paiva, "Pattern Based GUI Testing Modeling Environment", March 18, Fourth International Workshop on TESTing Techniques & Experimentation Benchmarks for Event-Driven Software - TESTBEDS, Co-located with The Sixth IEEE International Conference on Software Testing Verification and Validation, 2013

[27] Miguel Nabuco, Ana C. R. Paiva, "Model-based test case generation for Web Applications", 14th International Conference on Computational Science and Applications (ICCSA 2014), Guimaraes, Portugal, June 30 - July 3, 2014

[28] Miguel Nabuco, Ana C. R. Paiva, "Inferring User Interface Patterns from Execution Traces of Web Applications", Software Quality workshop of the 14th International Conference on Computational Science and Applications (ICCSA 2014), Guimaraes, Portugal, June 30 - July 3, 2014

[29] Paternò, Fabio, Cristiano Mancini, and Silvia Meniconi. "ConcurTask-Trees: A diagrammatic notation for specifying task models." Human-Computer Interaction INTERACT'97. Springer US, 1997.