Testing Android Incoming Calls

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

Marco A. Gonçalves
Faculty of Engineering of the University of Porto
Porto, Portugal
up201708897@fe.up.pt

André R. Barros
Faculty of Engineering of the University of Porto
Porto, Portugal
barros.andrer@gmail.com

***Abstract*—Mobile applications are increasingly present in our daily lives. Being increasingly dependent on apps, we all want to make sure apps work as expected.**

One way to increase confidence and quality of software is through testing. However, the existing approaches and tools still do not provide sufficient solutions for testing mobile apps with features different from the ones found in desktop or web applications. In particular, there are guidelines that mobile developers should follow and that may be tested automatically but, as far as we know, there are no tools that are able do it.

The iMPAcT tool combines exploration, reverse engineering and testing to check if mobile apps follow best practices to implement specific behavior called UI Patterns. Examples of UI Patterns within this catalog are: orientation, background-foreground, side drawer, tab-scroll, among others. For each of these behaviors (UI Patterns), the iMPAcT tool has a corresponding Test Pattern that checks if the UI Pattern implementation follows the guidelines.

This paper presents an extension to iMPAcT tool. It enables to test if Android apps work properly after receiving an incoming call, i.e., if the state of the screen after the call is the same as before getting the call.

It formalizes the problem, describes the overall approach, describes the architecture of the tool and reports an experiment performed over 61 public mobile apps.

***Keywords*-Mobile Testing; Android Testing; Software Testing; Software Test Automation; Pattern Based Testing**

I. Introduction

The smart phone industry has grown and is very competitive nowadays. Along with this, there is the development of more and more mobile apps, specially for the Android platform. As an increasingly important part of people's lives, it is important to work in order to increase the quality of those mobile apps.

One way to increase software quality is through testing, but testing mobile applications manually on the entire diversity of platforms and devices is not possible. Although there are already options for automated testing, which may be the easiest and cheapest way to perform these tests, most of these options are not enough.

Mobile applications have specific characteristics that are not found in Desktop or Web applications [20]. In particular, there are development guidelines that should be followed. Some of these best practices may be tested completely automatically. An illustrative example is the orientation change of a device. The state of the app before changing the screen from portrait to landscape and then again to portrait should be the same. However, there are lots of apps that do not implement this behavior properly. Also, this type of behavior may be tested automatically [25].

The iMPAcT tool combines exploration, reverse engineering and testing (Figure 5). The main goal is to test recurrent behavior (UI Patterns) on Android applications checking if the development guidelines are being followed. To test the identified UI Patterns, iMPAcT tool runs a set of corresponding test strategies (Test Patterns) over the Android application under test. The recurrent behavior to test and the test strategies to apply are part of a catalog of Patterns.

One aspect that we should not forget during mobile testing is that a mobile device is a mobile phone at its genesis. Being a mobile phone, it is noticeable that this will be in charge of receiving calls / messages. Taking into account that these devices run several different apps, we must ensure that the apps still work properly in the presence of incoming calls.

Besides being able to test mobile specific behavior, the iMPAcT tool is only able to test the behavior defined in the patterns' catalog. So, to test additional behavior related to incoming calls, we extended the catalog with the, so called, "Call Test Pattern".

This work presents a formalization of the problem theme of this work in Alloy in section II, then it describes the iMPAcT tool in section III. The Call Test Pattern is described in section IV, and its architecture is presented in section V. The study performed over 61 apps is described in section VI and threats to validity are in section VII. Section VIII presents related work and, finally, the conclusions are in section IX.

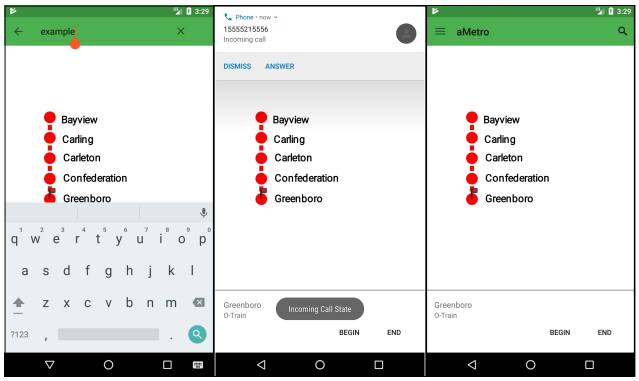

Figure 1. aMetro screen before/when/after receiving the call

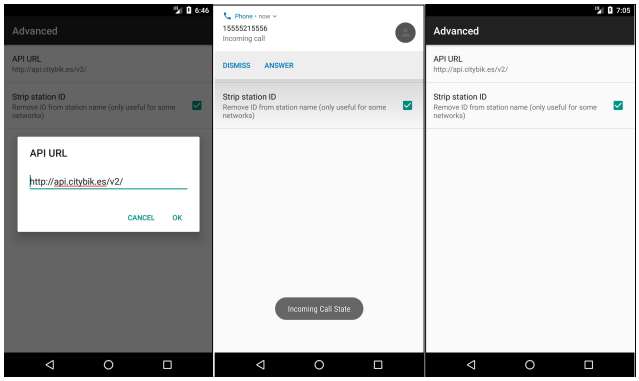

Figure 2. Openbike screen before/when/after receiving the call

II. Motivational Example

The number of apps available in the Google Play Store reached 3.3 million in March 2018[1]. Therefore, when one application does not behave as expected by users, another application is put quickly in its place [13].

Knowing and dealing properly with the activities life cycle is crucial to store the state of the apps when they stop running either because they are paused, stopped or destroyed. When this is not done correctly, users may get an incoming call in the middle of an interaction and loose the state after disconnecting the call.

An example of what may happen can be seen in Figures 1 and 2. In Figure 1 there is data inserted by the user on the *EditText* on the left image ("example") but after the call, the inserted text disappeared (image at the right side) (it shows "aMetro"). In Figure 2 there is a message box that disappears when getting an incoming call and is still missing when the call is finished.

Excluding the situation when the app crashes, the problems related to the incorrect handling of the app state can be classified as:

- properties of GUI elements changed
- new elements added to the screen
- screen elements disappeared

This type of problems may be detected automatically by inspecting the state of the app before and after the phone call. If the states are different, it means that the application, when receiving a phone call, does not maintain its previous state, which is a violation of the Android guidelines.

A. Formalization of the problems to detect

Alloy is a specification language that allows to describe structures and check, within a specific boundary, if some properties hold or not over a model. This language is supported by the tool, Alloy analyzer[2], which is a solver that takes the constraints of a model and finds structures that satisfy them.

Besides being useful to prove satisfability, Alloy analyzer can also be used to check properties of the model by looking for counterexamples. The structures generated (either valid instances or counterexamples) may be seen graphically, and their appearance can be customized for the domain at hand. So, we use Alloy to formalize the problem related to the wrong behaviour of the mobile apps when getting an incoming call.

First of all, we describe the static aspects of the problem, i.e., the structure/state of a mobile *Screen*. As such, a mobile GUI state may be seen as a tree of *ViewGroup* and *Views*.

A *ViewGroup* is an invisible container that defines the layout structure and that may contain other *ViewGroup* and *View*. A *View* is something the user can see and interact with. Each *View* within a *Screen* has a set of properties with some assigned *Value*. So, the overall state of a mobile screen (*Screen*) may be defined has the set of *ViewGroups*, a set of *Views*, and, for each *View*, a set of properties with assigned *Value*.

This static part of the problem may be described formally, using the Alloy specification language as follows.

```
sig View {}
sig ViewGroup {}
sig Value{}
sig Screen {
  structure: ViewGroup lone -> set ViewGroup,
  innerViews: ViewGroup lone -> set View,
  views: set View,
  properties: View some -> one Value
} {
  //last ViewGroup should have only Views
  all vg:structure[ViewGroup]+structure.ViewGroup
      | no structure[vg] => some innerViews[vg]
  // this struture should be acyclic
  all s1:structure[ViewGroup]+structure.ViewGroup
      | s1 not in s1.^structure
  // views of the first level cannot be inside
      another level
  no views & innerViews[ViewGroup]
}
run {}
```

[1]https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/

[2]http://alloytools.org/

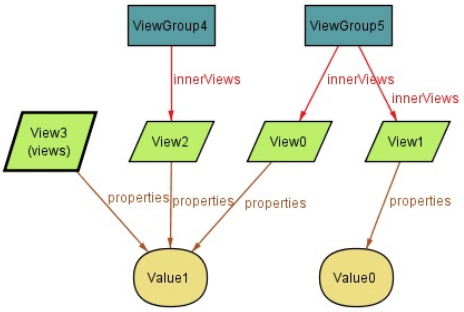

Figure 3. Screen structure instance/example

From this model, it is possible to generate instances that correspond to possible mobile screen states. One example may be seen in Figure 3 where a screen has *View3*, *ViewGroup4* and *ViewGroup5* at the first level and then, *ViewGroup4* has *View2* inside and *ViewGroup5* has *View0* and *View1* inside.

The dynamic part of the problem is related to the evolution of the Screen state along time. In particular, we aim to formalize the three types of problems that may occur when an app gets a Call and disconnects it. For that, we compare two consecutive screens (before getting a call and after turning off the call).

This dynamic aspects of the problem may also be described in Alloy, establishing an order between two states and defining three operations to describe the failures that may be found. In this case, these failures may be due to

- AddedView[s1,s2:Screen]
- RemovedView[s1,s2:Screen]
- DifferentProperty[s1,s2:Screen]

Formally, this can described in Alloy as follows:

```
/* establish a total order between Screen states*/
open util/ordering[Screen] as ord

/* to run this model, insert here the static part
    of the model present before*/

/* Added View in the final screen */
pred AddedView[s1,s2:Screen]{
  some v:View | v not in s1.views +
      s1.innerViews[ViewGroup] and v in s2.views +
      s2.innerViews[ViewGroup]
  and #(s1.views + s1.innerViews[ViewGroup]) <
      #(s2.views + s2.innerViews[ViewGroup])
  and CompareScreens[s1,s2,v]
}
/* Removed View in the final screen */
pred RemovedView[s1,s2:Screen]{
  some v:View | v in s1.views +
      s1.innerViews[ViewGroup] and v not in
      s2.views + s2.innerViews[ViewGroup]
  and #(s1.views + s1.innerViews[ViewGroup]) >
      #(s2.views + s2.innerViews[ViewGroup])
  and CompareScreens[s1,s2,v]
}
```

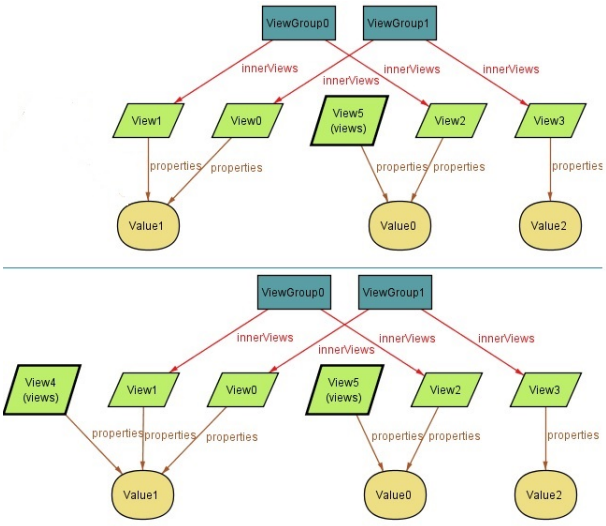

Figure 4. The screen at the bottom has an additional view (View4) that is not present on the screen at the top

```
/* Different View property in the final screen */
pred ChangedPropertyValue[s1,s2:Screen]{
  some v:View | s1.properties[v] !=
      s2.properties[v]
  and CompareScreens[s1,s2,v]
}
/* auxiliar predicate to compare two screen
    besides a change in View v */
pred CompareScreens[s1,s2:Screen, v:View]{
   s1.structure = s2.structure
   s1.innerViews :> {View- v} = s2.innerViews :>
       {View-v}
   (View-v) <: s1.properties = (View-v) <:
       s2.properties
   s1.views-v = s2.views-v
}
/* A problem should be detected when two screens,
    before and after the call, are different due
    to added View, removed View or changed
    property of a View */
fact {
  all s: Screen, s': ord/next[s] | AddedView[s,s']
      or RemovedView[s,s'] or
      ChangedPropertyValue[s,s']
}
run {} for 7
```

The formalization above will give us examples of two consecutive screen (before and after getting an incoming call) in which it is possible to detect one of the problems expressed though the fact. Either the screens vary because the apps during this process loose *Views*, gain *Views* or the existing *Views* change some of their properties. See an example in Figure 4 where *View4* is added to the final screen (screen on bottom).

III. iMPAcT Tool

The iMPAcT tool automates the testing of recurrent behavior (UI patterns) present on Android applications. This tool works in iterations combining automatic exploration,

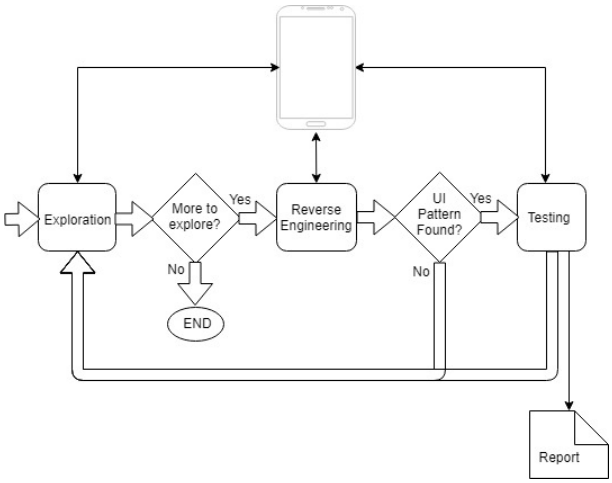

Figure 5. iMPAcT Tool Interative Process

inference of UI Patterns and testing the identified UI patterns (Figure 5).

The tool uses a predefined catalog of UI Patterns defined as <Goal, V, A, C, P>, where *Goal* is the *id* of the UI Pattern; *V* is a set of pairs (inputs of the UI Pattern and corresponding values); *A* is the sequence of actions to perform in order to infer the presence of the UI pattern; *C* is the set of checks to perform (if the checks pass it means that the pattern exists); *P* is the precondition that established the states where actions should occur trying to infer the presence of this UI pattern.

When the presence of a UI Pattern is detected, the iMPAcT tool runs the corresponding Test Pattern. A Test Pattern is formalized as a UI Pattern but the checks to perform will define if the test passes or fails, i.e., define if the corresponding UI Pattern is well implemented or not.

At this moment, the iMPAcT tool is able to infer and test several UI patterns: side drawer; orientation; tab-scroll; and background-foreground, among others. This catalog of patterns was defined based on the good practices for Android programming.

For example, the UI Orientation pattern checks if it is possible to rotate the screen. Whenever this is possible, the corresponding Test Pattern checks if the rotation is well implemented, i.e., if the user data inserted was not lost and if the main components of the screen are still present. The formalization of this pattern can be found in [4], [5].

In order to test the background-foreground pattern, the iMPAcT tool captures the information of the screen and then sends the app to background by pressing the home button. After that, the tool brings back the app to foreground by selecting it from the recent opened apps menu. Then, it captures again the information of the screen to compare with the previous one. If they are different, the test fails.

Besides being completely automatic, the iMPAcT tool is only able to test the UI patterns that belong to the catalog. So, whenever it finds the presence of a UI Pattern (described in the catalog) in the app under test, it applies the corresponding test strategy (test pattern within the catalog) defined. The process ends when the exploration process reaches the home page of the mobile device. This may happen because there is no more behavior to explore or because the tester presses the home button on purpose to stop the exploration process.

At this point in time, the iMPAcT tool is already able to compare two different screen states but it is not able to test the behaviour of the apps when getting and disconnecting a call while interacting with an app under test. This is called "Call Test Pattern", it is the contribution of this paper and is described in the next section.

IV. Call Test Pattern

It is important not to forget what an Android device is, at its genesis, a mobile phone. So, it will be able to get and make phone calls to other devices. However, being also a smart phone, means that the same device will be able to run different applications. Both aspects should work properly, so, when someone gets a call in the middle of an interaction, he should be able to return to the same state after hanging up the call.

We call this behavior check, the "Call Test Pattern". Every time there is an incoming call, this pattern hangs it up and, if the app did not crash, compares the state of the app before getting and after disconnecting the incoming call.

A. Call test pattern formalization

The call pattern is defined formally as follows:

- **Goal**: "Check if the app works properly after an incoming call"
- **P**: {"Incoming call"}
- **V**: {}
- **A**: [observe, hang up the call, observe]
- **C**: {"the app did not crash and the states of the app before and after the call are the same"}

B. Call Test Pattern process

The iMPAcT tool process followed to execute this pattern is different from the one described in Figure 5. In this particular case, the iMPAcT tool does not run the reverse engineering activity because there is not the need to infer if there is behavior to test as in other patterns. For instance, when we are testing if the side drawer is correctly implemented, the iMPAcT tool checks if the app has or not a side drawer. In this case, it is always possible to make/receive a Call so, there is no need to infer that and the iterative process is simplified having only exploration and testing activities. Also, there is the need for two emulators, one to make the call and another exploring the app under test (see Figures 6 and 7) and running the tests when getting a call.

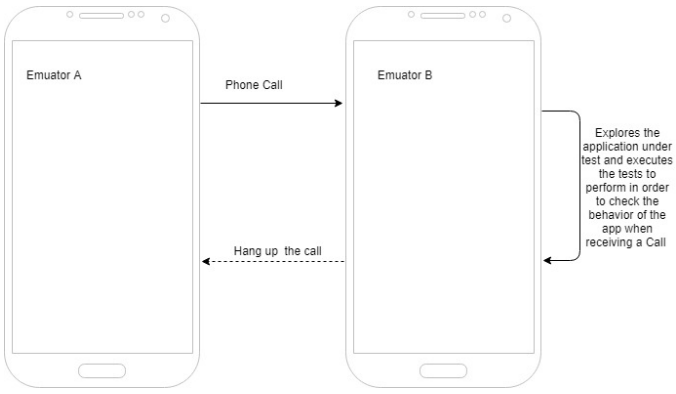

Figure 6. Call between two Emulators

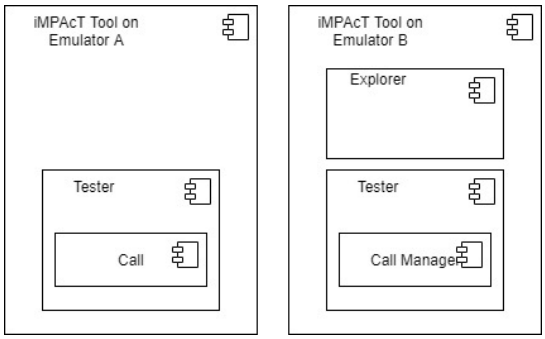

Figure 7. Architecture of iMPAcT tool in each Emulator

V. Call Test Pattern Architecture

To test the behavior of an app when receiving an incoming call, there is the need for two emulators. As can be seen in Figure 6, one emulator is responsible to make the calls (Emulator A) and the other (Emulator B) is responsible to disconnect the incoming call while exploring the app under test. The iMPAcT tool in Emulator A is responsible to make phone calls to Emulator B in fixed time intervals. The iMPAcT tool in Emulator B explores the app, captures the screen state before an incoming Call, disconnects the incoming Call and gets again the screen state after disconnecting the Call to compare with the previous state.

This is an iterative process that ends when the exploration process reaches the home page of the mobile device. This may happen because there is no more behavior to explore or because the tester presses, on purpose, the home button.

As such, Emulator A runs a "Call Activity" that is able to make calls to the other emulator, and Emulator B runs a "Call manager activity" that is able to detect an incoming call and to hung it up as described next.

A. Call activity

This activity starts in an emulator (emulator A in Figures 6 and 7) and is the only process that such emulator runs. This activity fires a call to the other emulator every 10 seconds. This time interval may be changed as desired. However, when executing the test pattern it seemed adequate regarding the time needed by the other emulator to detect and hang up the incoming call received.

B. Call manager activity

This component takes advantage of Android's ability to perform multiple tasks in parallel so that, unlike the previous component, it can run on the same device that is exploring automatically the app under test. This allows the iMPAcT tool to be able to detect and hang up incoming calls on the desired device, proceed with the execution of the corresponding test pattern and, afterwards, proceed with the execution of the app under test.

C. Reflection

"Relection" is the ability to run a program in order to make an analysis of itself and its environment and, afterwards, be able to changing its behavior based on these analyses. In order to do self-analysis, the program needs to have a representation of itself (*metadata*). In the case of an object-oriented programming environment (such as Java), the *metadata* is organized into objects called *metaobjects*. In general, there are three techniques that a reflection API can offer in order to change the behavior of the program [19]:

- direct modification of *metaobjects*;
- operations using *metadata* as dynamic function calls;
- intersection, where the code can intercede at various stages of program execution.

In order to access methods of the internal Android Telephony API, the call pattern uses a reflection API. This is done by defining the API representation as an AIDL (Android Interface Definition Language) file which, in practical terms, is a Java interface defining which Telephony methods to use. Having the API representation is now possible to invoke the methods needed to terminate the incoming call programmatically within the iMPAcT Tool.

At the end of an execution, iMPAcT tool reports the failures found and presents a FSM (Finite State Machine) where the nodes are screen traversed by the exploration and each arrow linking two nodes defines an event fired on the source screen that originated the target screen.

A video showing the tool working may be seen in youtube.com/watch?v=M2_qMSsOEIc&feature=youtu.be

VI. Exploratory study

This section lists the research question, describes the methodology followed by the exploratory study and explains how the subjects were selected. The study aims to answer the following research question:

RQ: Is it possible to find problems related to incoming calls in Android apps automatically?

In order to answer the aforementioned research question, we divided the study in three steps: **(1)** select the apps to

Table I
CRITERIA FOR APP SELECTION

Google Play Store	The application needs to be available in the Google Play Store
Google Play Store rating	The application's minimum rating is 3.5 to ensure it has some degree of quality
Google Play Store downloads	The application's minimum downloads is 10000 to ensure it has been used by multiple people
Use Gradle	The application must use Gradle to simplify the build of the application
Graphical User Interface	The application must have a Graphical User Interface to be tested by the iMPAcT tool
Western European Language	The application must be in an Western European Languages so its UI can be understood

Table II
APPLICATION'S CATEGORIES

Category	Number of Apps
Books and Reference	3
Communication	4
Education	2
Finance	1
Library and Demo	3
Lifestyle	1
Maps and Navigation	3
Medical	1
Music and Audio	4
News and Magazines	3
Photography	1
Productivity	8
Puzzle	1
Strategy	1
Tools	19
Travel and Local	4
Video Players and Editors	2
Total	**61**

analyze; **(2)** Execute the iMPAcT tool over the apps selected; **(3)** register and report the failures found.

The study was done on mobile applications that can be found in the Google Play Store. The final set of 61 apps was selected randomly from the ones obtained according to the criteria presented in Table I.

In addition we also analyzed the categories of the apps selected. The final set covers 17 categories as shown in (Table II).

A. Analysis of the results achieved

From the 61 apps tested, 11 revealed failures, which corresponds to 18% of the apps. Most of these apps with failures use GPS (7 apps) or editing files (3 apps) or editing images (1 app). Examples of the problems detected can be found in Figures 1 and 2.

The failures found may be explained by the fact that these applications do not have implemented mechanisms that deal with a sudden interruption of its work. That is, these applications when receiving a phone call, do not store its state properly.

There are several ways to save state on Android apps[3]. Some of the methods available to store data are: the use of *OnSaveInstanceState(Bundle)* method, the use of local persistent storage, and the use of *ViewModel*.

The *Bundle* populated by the *OnSaveInstanceState(Bundle)* method is passed to *onCreate(Bundle)* and *onRestoreInstanceState(Bundle)*. The *onSaveInstanceState(Bundle)* is used to store data that will be reloaded when the activity stops and is recreated.

Another way is to use persistent local storage, such as preferences or databases, to store all data which remains saved as long as the app is installed (unless the user deletes it on purpose).

Also, it is possible to use *ViewModel* [1] which retains data in memory. *ViewModel* is associated with an activity (or some other lifecycle owner). As such, it is scoped to the lifecycle and remains in memory until the scope of the lifecycle disappears permanently.

The problems detected by this test pattern should be reviewed later to decide if they are really issues that need to be fixed or if they are app-specific behavior that should be kept unchanged. Although we estimate that the failures found are real failures and due to problems related to the saving of state, it is not the goal of this paper to inspect each of the tested apps to identify the real faults within the code that give origin to these failures.

The goal of this work is to define an approach that is able to identify and point problems/failures that need to be analyzed and, eventually, fixed and the identification of the faults that give origin to these failures is left for future work.

VII. THREATS TO VALIDITY

Regarding the research question, we can conclude by the case study that it is possible to detect automatically problems related to receiving calls while interacting with a mobile app by using a fully automatic and black-box approach.

As such, it was possible to detect problems related to 11 of the 61 apps analyzed.

However, there are some threats to validity associated to the results we have presented.

External Validity. The overall study was performed over 61 apps. This set could be bigger, however, it should be noticed that the iMPAcT tool can take up to 20 minutes to test each app. In addition, we used four different exploration algorithms to test each app which may take up to 80 minutes per app tested. To mitigate this threat, we selected real-world apps of different categories to diversify the set of subjects used in the experiment.

Internal Validity. In terms of internal validity, our experiment may not be completely free of errors because of

[3]https://developer.android.com/

the nature of the iMPAcT tool used in the case study. The tool provides different exploration algorithms and the results achieved by this tool depend on the exploration algorithm used [21]. If the exploration does not reach the screen state where there is, for example, state to save before the incoming Call, the tool will not be able to detect the problem. Also, the Call Test Pattern makes calls every 10 seconds. So, it is not possible to guarantee that we are making calls where failures may be found. To mitigate this threat we have used all the different exploration methods supported by the iMPAcT tool which corresponds to 4 different executions per app. We tested each app using 4 different exploration methods and each exploration may take 20 minutes. So, we spent approximately 10 days with this experiment.

VIII. Related work

Mobile apps are part of our daily life. But, if an app does not correspond to the expectations of the user, it is common to abandon that app and install another one. So, it is of utmost importance to test them properly. However, mobile apps have specific characteristics and challenges not present on web or desktop applications, so, there is the need for specific testing tools [1] that are able to test those specificities, for instance, GPS, device orientation, and incoming messages and calls, among others.

There are different mobile testing tools. They may be classified according to different criteria [6]–[8], for instance, according to the testing activities they automate or according to the general testing strategy they use, such as black-box (do not have access to the app code) or white-box (have access to the app code).

According to the level of automation, we may split tools into the ones that automate the test case execution and others that are also able to automate the generation of test cases. Examples of tools that automate the test execution are the script-based tools. When using these tools, the testers write manually the test cases using APIs provided by test automation frameworks that allow interact automatically with the application under test (e.g., Robotium, Espresso). We may also classify Capture/Replay tools within this set [18]. Besides providing a mechanism to save the actions performed by the tester into a test script, they do not support the generation of test cases. The tester needs to design the tests and interact with the app under test in order to record the test script in a more easy way to replay such script later.

A different set of tools is the one that allows to generate test cases automatically. In this set we may find random testing tools and model-based testing tools. Random testing tools perform randomly actions over the app under test. These tools may find failures that provoke crashes [3]. The other example of this type of tools are the model-based ones. In this case, the tools design test cases based on models of the app under test [16], [24]. The models may vary in nature [9]–[12] and according to that, different test case generation techniques may be used. The main problem with these tools is the effort needed to build the model. To diminish this effort, some tools try to infer or extract a model [16], [17], [22] and others have a predefined model which describes common traversal behavior to test over all mobile apps [4], [5], [14], [15].

According to the general test strategy, we may classify mobile testing tools as black-box (this is the case of random tools) and white-box (that are more common at the unit testing level).

Black-box tools explore dynamically the apps under test [2]. These tools vary on the exploration algorithm used: it can be completely random, following some heuristics [21] or guided defining the next action to perform based on the knowledge acquired from the exploration performed until a specific moment (Active Learning). Also, the exploration process may be used as a reverse engineering process to extract information about the app being tested, such as screens states, screen images, and available events.

Within the set of white box testing tools we may find static analysis tools [23] that parse the source code (or bytecode) of the apps checking some properties. This type of tools may be used for different purposes: assessing the security, detecting app clones, automating test cases generation, or for uncovering non-functional issues related to performance or energy [23].

However, as far as we know, there is no tool that tests the behavior of the mobile apps when receiving and hanging up an incoming call.

IX. Conclusions

While interacting with a mobile app, it is possible to get an incoming call. However, not always the apps are prepared properly to deal with that.

This paper presents a testing tool that is able to test the behavior of Android apps when getting and discarding an incoming call. The goal is to check if the screens before and after an incoming call are the same, i.e., no new elements, no removed elements and no changes in existing elements. This tool is implemented as an extension to iMPAcT tool that is able to test if the behavior of mobile apps described in a pattern's catalog is correct or not. This research work adds a new pattern to that catalog, the so called, "Call Test Pattern".

The architecture and the process followed by this Call Test Pattern are presented. In addition, we present an experiment performed over 61 apps. In 11 of the 61 apps analyzed, it was possible to detect potential problems.

Currently, calls are triggered at a fixed time interval and failures are detected when screens are different. In the future, we intend to fire the calls at random intervals and implement a boundary-based approach to point out failures.

Overall, from the experiments performed, we can conclude that it is possible to detect automatically problems

related to getting and discarding incoming calls. It is our believe that this is a useful tool to help improving the quality of Android apps.

REFERENCES

[1] Henry Muccini, Antonio Di Francesco, and Patrizio Esposito, "Software testing of mobile applications: challenges and future research directions", In Proceedings of the 7th International Workshop on Automation of Software Test (AST '12). IEEE Press, Piscataway, NJ, USA, 29-35, 2012.

[2] X. Ma, N. Wang, P. Xie, J. Zhou, X. Zhang, C. Fang, "An Automated Testing Platform for Mobile Applications", in 2016 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C), pp.159-162, 1-3 Aug, 2016.

[3] Moran, K., Linares-Vasquez, M., Bernal-Cardenas, C., Vendome, C., Poshyvanyk, D.. "Automatically discovering, reporting and reproducing android application crashes", in IEEE International Conference on Software Testing, Verification and Validation (ICST), IEEE, Chicago, IL, USA, pp 3344, 2016.

[4] Inês Coimbra Morgado; Ana C. R. Paiva, "The iMPAcT Tool: Testing UI Patterns on Mobile Applications", 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp.876-881, 2015.

[5] Inês Coimbra Morgado; Ana C. R. Paiva, "Testing Approach for Mobile Applications through Reverse Engineering of UI Patterns", 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW), pp.42-49, 2015.

[6] Nurul Husna Saad; Normi Sham Awang Abu Bakar, "Automated testing tools for mobile applications", The 5th International Conference on Information and Communication Technology for The Muslim World (ICT4M), pp.1-5, 2014.

[7] J. Gao, X. Bai, W. T. Tsai, T. Uehara, "Mobile Application Testing: A Tutorial", in Computer, Vol.47, Feb. 2014.

[8] D. Amalfitano, N. Amatuccia, A. Memon, P. Tramontana, A.R. Fasolino, "A general framework for comparing automatic testing techniques of Android mobile apps", in Journal of Systems and Software, Vol.125, pp. 322–343, March 2017.

[9] Nguyen, C.D., Marchetto, A., Tonella, P., "Combining model-based and combinatorial testing for effective test case generation", in Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2012), ACM Press, Minneapolis, MN, USA, pp 100110, 2012.

[10] Franke, D., Kowalewski, S., Weise, C., Prakobkosol, N., "Testing conformance of life cycle dependent properties of mobile applications", in the IEEE Fifth International Conference on Software Testing, Verification and Validation, IEEE, Montreal, QC, Canada, pp 241250, 2012.

[11] Avancini, A., Ceccato, M., "Security testing of the communication among android applications", in Automation of Software Test (AST), 2013 8th International Workshop on, IEEE Press, San Francisco, California, pp 5763, 2013.

[12] Gorla, A., Tavecchia, I., Gross, F., Zeller, A., "Checking app behavior against app descriptions", in Proceedings of the 36th International Conference on Software Engineering (ICSE 2014), ACM Press, Hyderabad, India, pp 10251035, 2014.

[13] Localytics Announces Industry Benchmarks for App Engagement and User Retention, Boston, 13 March, 2017.

[14] Inês Coimbra Morgado, Ana C. R. Paiva, "Test Patterns for Android Mobile Applications", in EuroPLoP - 20th European Conference on Pattern Languages of Programs, 2015.

[15] Inês Coimbra Morgado, Ana C. R. Paiva, João Pascoal Faria, "Automated Pattern-Based Testing of Mobile Applications", in 5th Portuguese Software Engineering Doctoral Symposium (SEDES 2014) hosted by 9th International Conference on the Quality of Information and Communications Technology (QUATIC'14), 2014.

[16] D. Amalfitano, A.R. Fasolino, P. Tramontana, B.D. Ta, A.M. Memon, "Mobiguitar: automated model-based testing of mobile apps", IEEE Softw., 32 (5), pp. 53-59, 2015.

[17] R. Mahmood, N. Mirzaei, S. Malek, "Evodroid: Segmented evolutionary testing of android apps", Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, New York, NY, USA, pp. 599-609, 2014.

[18] C. H. Liu, C. Y. Lu, S. J. Cheng, K. Y. Chang, Y. C. Hsiao and W. M. Chu, "Capture-Replay Testing for Android Applications", in International Symposium on Computer, Consumer and Control, 2014.

[19] Ira R. Forman and Nate Forman and Dr. John Vlissides Ibm and Ira R. Forman and Nate Forman, "Java Reflection in Action", 2004.

[20] Inês Coimbra Morgado, Ana C. R. Paiva, "Mobile GUI testing", in Software Quality Journal, vol.26, n.4, pp.1553–1570, 2018.

[21] Inês Coimbra Morgado, Ana C. R. Paiva, "Impact of execution modes on finding Android failures", in Procedia Computer Science, vol.83, pp.284-291, ISSN 1877-0509, 2016.

[22] Inês Coimbra Morgado, Ana C. R. Paiva, "Testing approach for mobile applications through reverse engineering of UI patterns", in TESTBEDS - Sixth International Workshop on Testing Techniques for Event BasED Software, 2015.

[23] Li, Li and Bissyand, Tegawend F. and Papadakis, Mike and Rasthofer, Siegfried and Bartel, Alexandre and Octeau, Damien and Klein, Jacques and Traon, Le, "Static Analysis of Android Apps", in Inf. Softw. Technol. Journal, vol.88, n.C, 2017.

[24] Pedro Costa, Ana C. R. Paiva, and Miguel Nabuco, "Pattern Based GUI Testing for Mobile Applications", in 9th International Conference on the Quality of Information and Communications Technology (QUATIC'14), 2014.

[25] Domenico Amalfitano, Vincenzo Riccio, Ana C. R. Paiva, Anna Rita Fasolino, "Why does the orientation change mess up my Android application? From GUI failures to code faults", in Journal of Software: Testing, Verification and Reliability (STVR), 2017.