Testing When Mobile Apps Go to Background and Come Back to Foreground

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

João M. E. P. Gouveia
Faculty of Engineering, University of Porto
Porto, Portugal
up201303988@fe.up.pt

Jean-David Elizabeth
Higher Institute for Electronics and Digital Training
Brest, France
jean-david.elizabeth@isen-ouest.yncrea.fr

Márcio E. Delamaro
Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo
São Paulo, Brasil
delamaro@icmc.usp.br

***Abstract*—Mobile applications have some specific characteristics not found on web and desktop applications. The mobile testing tools available may not be prepared to detect problems related to those specificities. So, it is important to assess the quality of the test cases generated/executed by mobile testing tools in order to check if they are able to find those specific problems.**

One way to assess the quality of a test suite is through mutation testing. This paper presents new mutation operators created to inject faults leading to known failures related to the non-preservation of users transient UI state when mobile applications go to background and then come back to foreground. A set of mutation operators is presented and the rational behind its construction is explained.

A case study illustrates the approach to evaluate a mobile testing tool. In this study, the tool used is called iMPAcT tool, however any other mobile testing tool could be used. The experiments are performed over mobile applications publicly available on the Google Play store. The results are presented and discussed.

Finally, some improvements are suggested for the iMPAcT tool in order to be able to generate test cases that can kill more mutants and so, hopefully, detect more failures in the future.

***Keywords*-Mutation Testing; Mutation Operators; Mobile Testing; Android Testing; Software Testing; Software Test Automation**

I. Introduction

According to [11], 70 percent of software interactions in enterprises will occur on mobile devices, by 2022. So, it is of utmost importance to ensure the quality of mobile apps in all its facets (e.g., functionality, usability, security).

One way to increase the quality of software is through testing. However, testing mobile applications has additional challenges [26]. Mobile apps get external events (e.g., incoming calls and messages), use built in sensors (e.g., GPS, accelerometer), have specific activity states (e.g., paused, resumed, stopped), different ways of interaction (e.g., long press, swipe), and may run in a myriad of devices and different platforms (e.g., Android, iOS). So, in this context, it may be difficult to build a good test suite of a manageable size that is able to deal with all these specificities.

Mutation testing can be used to assess the quality of a test suite and/or to guide the generation of test cases. Mutation testing injects intentionally code changes and attempts to find the possible failures generated by those faults using existing test cases. One advantage of mutation testing is that it can be adapted to different contexts. Besides programs in several languages like Fortran, C, Python and many others, mutation testing has been used in artifacts like Finite State Machines [17] and Petri Nets [18]. In addition, mutation operators can be designed to address specific types of faults [9].

Although traditional operators designed for programming languages may not be enough to test particular characteristics of mobile applications, it is always possible to augment the set of existing operators with some specific to the application domain and/or technology. Examples of such research works are the papers of Linares-Vsquez et al. [13] and Moran et al. [14].

This paper presents additional mutation operators to test the specific behavior of mobile applications related to the non-preservation of users transient UI state (e.g., text in an *EditText* widget or the scroll position of a *ListView* widget) when apps are sent to background and then back to foreground.

In the next section a motivational example illustrating the problem is presented; the mutation operators defined are described in Section III; Section IV describes the exploratory study performed; Section V discusses threats to validity; the related work is presented in Section VI; finally, conclusions and future work are in Section VII.

II. Motivational Example

When an application is sent to background and then returns back to foreground, unless some computing process

has been performed, the user expects the application to return to the same state in which it was sent to the background. However, this is not always the case.

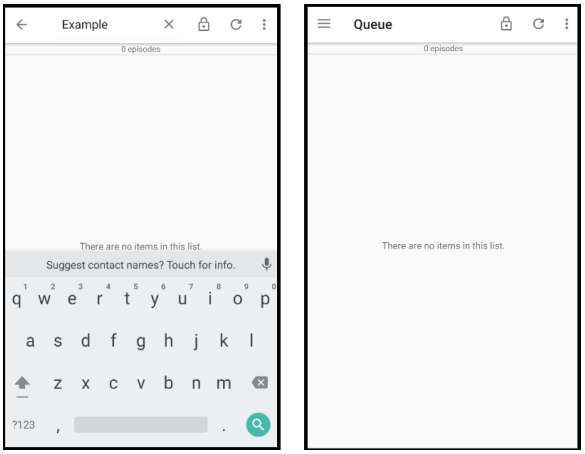

Figure 1. Example where the control showing the keyboard disappears and the word on top changed from "Example" to "Queue".

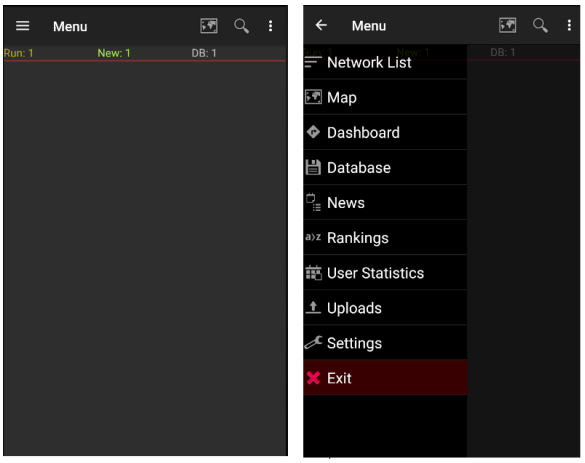

Figure 2. Example where a control (Side Drawer) appears when the app comes back to foreground.

There are cases where the state of the app changes (e.g., Figure 1 in which the word on top changes from "Example" to "Queue"), others in which new widgets come up (e.g., Figure 2 with the sidebar opening without user explicit intention) and other situations in which widgets disappear (e.g., Figure 3 where the available tools are not shown anymore). Such problems may disappoint the user and cause her/him to give up using an app. So, there is the need to build test suites that are able to detect these problems.

Mutation testing is a fault based testing technique that may be used to guide the generation of test cases and to assess the quality of a test suite. The test suite is run on the original application and over a sightly different

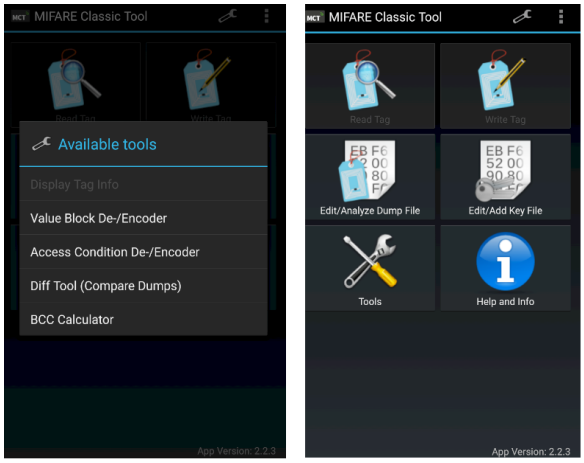

Figure 3. Example where a control showing the list of available tools disappeared after the app comes back to foreground.

version of the same application. The modified version is obtained by applying a mutation operator. Since mutation operators try to mimic real failures, a good test suite will be able to distinguish the original from the altered version of the application. One advantage of mutation testing is that mutation operators can be designed to address specific types of failures, like the ones mentioned in this section.

Next section describes the mutation operators defined to mimic the type of the failures illustrated in Figures 1, 2 and 3. The mutation operators were defined according to the following sequence of steps: (1) Analyze/study what happens with the lifecycle of Android apps when they are sent to background and then back to foreground. (2) Analyze/study the possible ways a developer may use to preserve users transient UI state. (3) Define the set of mutation operators. (4) Validate the mutation operators over some mobile apps to check if they are able to perform the expected changes.

III. Mutation Operators

In order to define mutation operators that can mimic failures related to the preservation of users UI transient state, there is the need to understand the Android activities' lifecycle and know the existing methods programmers can use to preserve such state.

A. Android activities' lifecycle

The Activity class is a core component of an Android app. Unlike other systems, activity instances are the place where Android systems start the code invoking specific callback methods that are related to specific stages of the lifecycle (Figure 4). Over the course of its lifetime, an activity goes through a number of states. So, in order to navigate through the transitions between the lifecycle stages of the activity, the Activity class provides six callbacks [22]:

onCreate(), *onStart()*, *onResume()*, *onPause()*, *onStop()*, and *onDestroy()*. The system invokes each of these callbacks when an activity enters a new state.

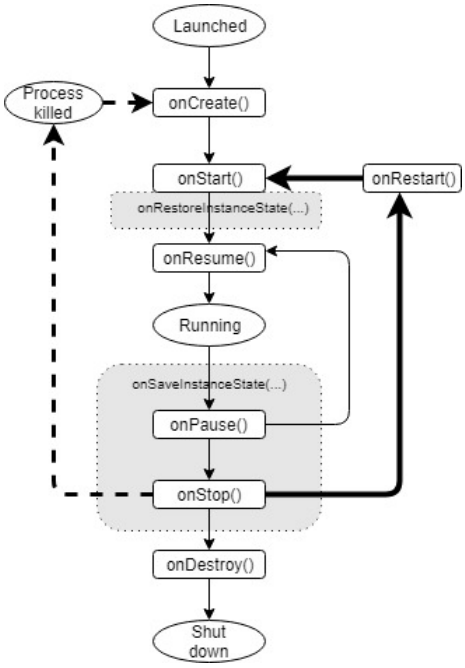

Figure 4. Android lifecycle (based on [22]).

Method *onCreate()* fires when the system first creates the activity. After the *onCreate()* method finishes execution, the activity enters the *Started* state, and the system calls the *onStart()* method. The *onStart()* method call makes the activity visible to the user, as the app prepares for the activity to enter the foreground and become interactive. Once this callback finishes, the activity enters the *Resumed* state. When the activity enters the *Resumed* state, it comes to the foreground, and then the system invokes the *onResume()* callback. This is when the app is running and interacting with the user. When the activity is running and an interruptive event occurs, the activity enters the *Paused* state, and the system invokes the *onPause()* callback. After completion of the *onPause()*, the activity remains in this state until either the activity resumes, the app process is killed because apps with higher priority need memory, or becomes completely invisible to the user. In the latter case, the system calls *onStop()*. From the *Stopped* state, the activity either comes back to interact with the user, or the activity is finished running and goes away. If the activity comes back, the system invokes *onRestart()*. If the activity is finished running, the system calls *onDestroy()* that is called before the activity is destroyed.

In order to define the mutation operators, we analyzed what happens within the lifecycle when an app is sent to background and then back to foreground.

When an application goes to background and becomes invisible to the user, the system invokes *onPause()* and *onStop()*. After these methods, and excluding the case in which the app is destroyed, at least two different sequences of method calls may happen: (1) the user navigates back to the activity in which case *onRestart()*, *onStart()* and *onResume()* are called (arrow in bold in Figure 4); (2) apps with higher priority need memory and, in this case, *onCreate()*, *onStart()*, and *onResume()* are called (arrow in bold dashed in Figure 4).

The Android framework is responsible to manage the lifecycle of the UI controllers (such as activities) and it is impossible to know when the framework will destroy or recreate them because this decision depends on user actions and device events. In case the activity is destroyed or recreated, the UI's transient state is lost [1].

B. How to preserve user's transient UI state

In order to make sure that an app comes back to foreground in the same state it was before sent to background, it is necessary a sequence of methods call that preserves its transient UI state. To do so, the developers may (among other options) [3]:

Save instance state by using the *onSaveInstanceState(Bundle)* method. The *Bundle* populated by this method is passed to *onCreate(Bundle)* and *onRestoreInstanceState(Bundle)* that allows to restore the state. The *onSaveInstanceState(Bundle)* is used to store data that will be reloaded when the activity stops and is recreated.

Use persistent local storage, such as preferences or databases, to store all data that has to be preserved if the activity is open and closed. This data remains saved as long as the app is installed (unless the user deletes it on purpose).

Retain data in memory by using ViewModel [1]. ViewModel retains data in memory and 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.

C. Definition of the mutation operators

In order to define the mutation operators, it is necessary to define:

- which failures we want to produce;
- which faults we can introduce to induce such failures.

The first are the problems created when an app comes to foreground, after being sent to background, illustrated in Figures 1, 2, 3. The second are faults that are related to the different forms available to developers to preserve the transient state of the user interface.

1) Mutation Operator 1 – CBDL: The *onSaveInstanceState(Bundle)* method is called when the activity begins to stop, i.e., when the activity may temporarily be destroyed. For some targeting platforms, this method is called after *onStop()*. For other platforms, this method is called before

onStop() either before or after *onPause()* [22] (grey zone around *onSaveInstanceState(Bundle)* in Figure 4). However, there are situations in which this method may not be called.

The first mutation operator (CBDL) was built based on the situation when the app overrides the *OnSaveInstanceState(Bundle)* method of an activity. This is helpful when developers need to save additional instance state information for the activity. The added key-value pairs to the *Bundle* object are saved in case the activity is destroyed unexpectedly.

Most apps use *onCreate(Bundle)* to restore the state, but sometimes they use the *onRestoreInstanceState(Bundle)*. This method may be called after *onStart()* (grey zone around *onRestoreInstanceState()* in Figure 4) and its default implementation restores the state previously saved by *onSaveInstanceState(Bundle)*.

The mutation operator clears the *Bundle* which contains the data to be restored in the future. Consider the following example where the value of variable "v" is being saved and restored afterwards in the *onRestoreInstanceState(Bundle)* method.

```
public class DataEntryActivity extends Activity {
   private ExClass v = ExClass.getInstance();
   // ... the rest of the activity
}
@Override
protected void onSaveInstanceState(Bundle
    outState) {
   super.onSaveInstanceState(outState);
   outState.putSerializable("v", v);
}
@Override
protected void onRestoreInstanceState(Bundle
    savedState) {
   super.onRestoreInstanceState(savedState);
   v = (ExClass)
       savedInstanceState.getSerializable("v");
}
```

In this case, the mutation operator updates the body of *onSaveInstanceSate(Bundle)* by adding a line that clears the *outState* Bundle.

```
(Mutant 1: CBDL)
...
@Override
protected void onSaveInstanceState(Bundle
    outState) {
   super.onSaveInstanceState(outState);
   outState.putSerializable("v", v);
   outState.clear();
}
...
```

If the app calls *onRestoreInstanceState(Bundle)* or *onCreate()* before coming back to foreground, it will restore an empty *Bundle* which is not the expected behavior and situations like the one presented in Figure 1 may happen.

2) Mutation Operator 2 – CPSE: This mutation operator is applied when developers persist data in local storage using shared preferences.

```
...
SharedPreferences v0 =
    getSharedPreferences(file_name, mode);
...
```

When that happens, this mutation operator clears and commits the empty editor in the *onPause()* method.

```
(Mutant 2: CPSE)

public void onPause() {
   super.onPause();
   ...
   getSharedPreferences(file_name,
       mode).edit().clear().commit();
}
```

So, when the app tries to read shared preferences again to get the saved state, it will read an empty shared preferences and, again, situations illustrated in Figure 1 may happen.

3) Mutation Operator 3 – NACT: When an activity B appears in the foreground taking focus and completely covering another activity A, activity A loses focus and enters the *Stopped* state. The system calls *onPause()* and *onStop()*. When the same instance of activity A comes back to foreground, the system calls *onRestart()*, *onStart()* and *onResume()*.

The third mutation operator (NACT) updates the *onPause()* overridden method (or creates one if it does not exist). It creates an intent (i.e., an abstract description of an operation to be performed [6]) to other activity of the app and starts such activity inside the *onPause()* method.

```
@Override
 protected void onPause() {
    super.onPause();
    ...
}
```

This will start another activity when the app is paused and, as such, the new activity takes the focus when coming back to foreground. This mutation operator can mimic situations where controls disappear and/or new controls appear and that are illustrated in Figures 2 and 3.

```
(Mutant 3: NACT)
@Override
protected void onPause() {
   super.onPause();
   ...
   android.content.Intent intentMutant = new
       android.content.Intent(this,
       OtherActivity.class);
   startActivity(intentMutant);
 }
```

In order to use mutant 3, the word "OtherActivity" should be replaced by another activity of the app being mutated. So, the developer should access all the activities of the app and select one of those randomly.

A black-box testing tool able to detect these problems has to compare the state of the screen before sending the app to background with the state of the screen after getting back the app to foreground.

D. Validating the mutation operators

The last step of our work is described in the next section. The mutation operators have been validated in an experiment with actual apps obtained from the Google PlayStore.

IV. Exploratory study

This section lists the research questions, describes the methodology followed and explains how the objects (programs) were selected. In summary, this study aims to assess the fault-detection effectiveness of the mutation operators defined and answer the following research question:

RQ: *Is it possible to use these mutants to measure the quality of a test suite (or testing tool) to identify problems of background & foreground transition?*

In order to answer the aforementioned research question, we divided the study in 5 steps:

1: select the objects for the study;
2: analyze the apps manually to check if the background & foreground behavior was well implemented;
3: apply the mutation operators over the correct apps;
4: analyze manually the failures caused by injected faults;
5: assess the quality of the test cases generated by a testing tool. In this experiment we have used the iMPAcT Tool that is described in the sequel but any other mobile testing tool or test suite could be used for this purpose.

A. iMPAcT Tool

The iMPAcT tool automates the testing of recurrent behavior (UI patterns) present on Android applications [4], [5], [19]–[21], [23]. This tool works in iterations combining automatic exploration, reverse engineering [24] (i.e., inference of UI Patterns) and testing the identified UI patterns using predefined test strategies called Test Patterns (see 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); and *P* is the precondition that established the states where actions should occur trying to infer the presence of this UI pattern.

When 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 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 whether 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 the main components of the screen are still present. This pattern is described in [4], [5].

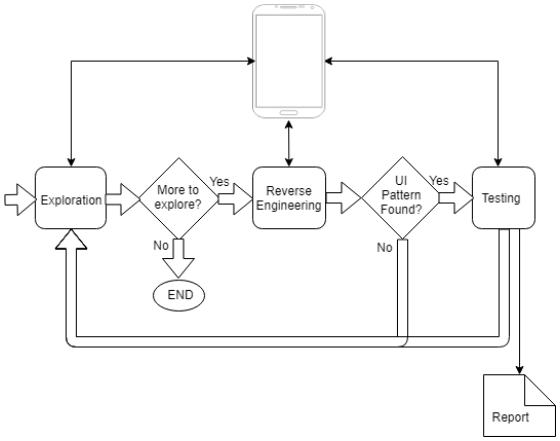

Figure 5. The iMPAcT tool testing process.

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, either because the set of widgets is not the same or because values of the widgets are different, the test fails.

The aim of using the iMPAcT tool in this experiment is to check if the background & foreground test strategy defined and used by the tool is capable of finding the problems related to not following the guidelines for implementing this behavior. If the test suite produced by the tool kills the mutants, we can suppose that it is complete regarding the testing of background & foreground transitions. Otherwise, it is incomplete.

As mentioned before, the goal of this experiment is to evaluate the mutation operators, so the choice of using the iMPAcT tool is a matter of convenience for the authors.

B. Objects

The study was performed on mobile apps that can be found in the Google Play Store and according to the criteria presented in Table I. From this set of apps we selected randomly 56 for our study.

In addition, we also analyzed the categories of the apps selected. The final set covers 18 categories as shown in (Table II).

Table I
CRITERIA FOR APP SELECTION

Google Play Store	The app needs to be available in the Google Play Store
Open source	The app code must be available to allow analysis and insert mutants
Google Play Store rating	The app's minimum rating is 3.5 to ensure some degree of quality
Google Play Store downloads	The app's minimum downloads is 10000 to ensure it has been used by multiple people
Android Native	The app code must be native Android because this is the context of this work
Graphical User Interface	The app must have a Graphical User Interface to be tested by the iMPAcT tool
Western European Language	The app must be in an Western European Languages so its User Interface can be understood

Table II
APPLICATION'S CATEGORIES

Category	Number of apps
Books and Reference	2
Communication	3
Education	2
Finance	1
Library and Demo	3
Lifestyle	1
Maps and Navigation	4
Medical	2
Music and Audio	3
News and Magazines	2
Photography	2
Productivity	6
Puzzle	1
Health and fitness	1
Strategy	1
Tools	16
Travel and Local	4
Video Players and Editors	2
Total	**56**

C. Manually Check of the Background & Foreground Behavior

The 56 apps were explored manually in order to check if there were failures related to background & foreground behavior. We could detect failures in 6 of them. The failures found on the six applications can be classified according to different types:

- State change error;
- Widget appearing;
- Widget disappearing.

D. Apply Mutation Operators

The mutation operators were applied over the 50 apps (the ones classified manually as correct). The results of that process is visible in Table III.

Table III
NUMBER OF APPS FOR WHICH MUTANTS WERE GENERATED AND NUMBER OF MUTANTS GENERATED

	Apps	Mutants Generated
Mutation Operator 1 (CBDL)	22	112
Mutation Operator 2 (CPSE)	14	33
Mutation Operator 3 (NACT)	44	466
Total	**80**	**611**

Mutation operator 3 (NACT) is the one that generates the higher number of mutants. In principle, this mutation operator can be applied in all the apps used in the case study because it updates the *onPause()* method or creates one if it does not exist. However, for 6 apps, the mutation operator generated mutants that could not be built and run. This happened because it directs the app to another activity but when the app has only one activity, this mutation operator does not work correctly. So, it was applied over 44 apps as can be seen in Table III.

Mutation operator 1 (CBDL) was applied over 22 different apps. The other 38 apps of the set did not use the *onSaveInstanceState(Bundle)* method.

Mutation operator 2 (CPSE) was applied over 14 apps. In the other 36 apps, it was not possible to find the use of shared preferences.

It should be noticed that we could apply more than one mutation operator over some apps. That is the reason why the sum of the apps from which we could generate mutants is higher than 50 in Table III.

E. Assess the Quality of the iMPAcT Tool Tests

The 611 mutants generated were executed by the iMPAcT tool. We installed each mutated version of the app in the device (either emulator or real device) and run the iMPAcT tool. The applications were compiled using Android Studio 3.1 and ran both in an emulator (Nexus 6 with Android 7.0 and Nexus S with Android 7.0) and a real device (Moto E with Android 6.0).

All mutants that did not build/compile correctly were discarded as this would disable testing the UI of the application in question. The most common errors building/compiling the applications were: missing google-services.json file; Gradle sync failed due to missing files; generic build failed error message due to missing files.

The results obtained were divided in 4 categories:

- **Not Spotted** — The failure was not detected neither manually nor via the *iMPAcT tool*.
- **Not Detected** — The failure was detected manually but not via the *iMPAcT tool*.
- **Detected** — The failure was detected both manually and via the *iMPAcT tool*.
- **Incorrect** — The injection of the mutant causes the application to crash or not to build.

Table IV
MUTATED APPLICATION: MANUAL AND *iMPAcT tool* TESTING RESULTS

	Operator 1	Operator 2	Operator 3
Not Spotted	28	27	0
Not Detected	6	1	0
Detected	78	5	457
Incorrect	0	0	9
Total	112	33	466

1) ***Mutation Operator 1 – CBDL****:* It was possible to generate 112 mutants from which 28 were "Not Spotted"; 6 were "Not Detected"; and 78 were "Detected".

The 28 mutants (generated from 6 apps) classified as "Not spotted" may be due to the generation of equivalent mutants, one of the problems related to practical usage of mutation testing. Two mutants are considered equivalent when they always behave as the original app. In this case it means that both the original app and its mutated version are considered correct with respect to the background & foreground behavior. This may happen because these apps override the *onRestoreInstanceState(Bundle)* method but the method's body is empty. Since it does not call the super-class method neither reads the Bundle, the behavior of the app is the same regarding background & foreground behavior despite the change in the *onSaveInstanceState()* method.

There are 6 mutants (generated from 2 apps) that could be killed manually but not with the iMPAcT tool. The reasons may be related to the exploration process used by the tool. Although the iMPAcT tool has different exploration algorithms, the behavior affected by the mutation operator may not be explored automatically because the exploration process does not reach that point of the app; or because the failures detected manually are not detected by the iMPAcT tool, e.g., color changes.

All the other 78 mutants (generated from 14 different apps) were detected manually and by the iMPAcT tool. So, we could check manually the incorrect behavior of the mutated versions of the apps and the iMPAcT tool could distinguish the screens of the apps before going to background and after coming back to foreground.

2) ***Mutation Operator 2 – CPSE****:* Shared preferences do not have impact over background & foreground often. They are generally used to save data when the activity is closed (not on *onPause()*). Often, apps read the preferences on *onCreate()* not on *onResume()*. However, we could find shared preferences impacting over background & foreground behavior in 3 apps.

In one of them, there is one mutant in which we could find the problem manually but not automatically. When running the mutant we can insert data in an edit text box. After moving the app to background and then back to foreground, the input data is lost and the text box returns with the default value, so, the input data inserted before was lost. However, this mutant could not be killed by the iMPAcT tool. We noticed that the edit text control used by the app is not a standard control. It is a specific class developed by the programmers. So, maybe the iMPAcT tool is not able to get the state of this special control and, as a consequence, cannot compare it with the state before going to background.

There are two other apps (5 mutants) in which we could kill the mutant manually and with the iMPAcT tool. When running these mutants we can configure a set of preferences. When the mutants go to background and then return to foreground, the configurations that were set are lost and turned to default values. This happens because, in these apps, the preferences are read in the *onResume()* method. The *onResume()* method is called when the mutants come back to foreground, so the empty preferences are read and their values become the default values again.

Regarding the other 11 apps, we could generate 27 mutants. These mutants could not be killed manually nor by the iMPAcT tool. There are 2 different explanations for this to happen.

The first case happens with 10 apps. The preferences are used to save data when closing the app. These preferences are read when opening the app again, i.e., when calling *onCreate()* on the main activity. In order to find the behavior change manually, we had to leave the activity with the back button and then open the activity again. In this case the methods called are *onCreate()*, *onStart()* and *onResume()*. But when we test these apps with the iMPAcT tool, the app is sent to background and then back to foreground which may be calling the methods *onRestart()*, *onStart()* and *onResume()* without calling the *onCreate()*. In these apps we can say that the mutants created are similar to the original ones with respect to the background & foreground behavior.

The second case happens with one app (3 mutants). Preferences are saved when the app is closed and the widget of the app on the home page changes accordingly. This is not detected by the iMPAcT tool because it does not close the app and opens the app again when it tests the background & foreground behavior. Even if it did that, close and open the app again, it would not kill the mutant because the iMPAcT tool does not compare the widgets of the apps on the home page.

3) ***Mutation Operator 3 – NACT****:* There are 457 mutants that were killed both manually and by the iMPAcT tool. When a new activity starts within the *onPause()* method, and the app comes back to the foreground again it returns in a new activity and so, the states of the screens before going to background and after coming back to foreground are different.

There are 9 "Incorrect" mutant apps that crashed while being tested. This mutation operator is the one which causes more disturbance in the background & foreground behavior by sending the app to a different activity. If before entering an activity, the application needs to have some conditions met and this does not happen and, if the error handling is

not correctly implemented, the application will likely crash. For example, if the application tries to open an activity which needs to have the user logged in and this condition is not met, the application will crash without the proper error handling. This might be the reason for the number of "Incorrect" mutant apps, which happened all in the same application.

F. Analysis of the results achieved

Regarding the research questions, we can conclude from the case study that it is possible to generate these 3 classes of mutants and that it is possible to apply them on real apps (real apps available on the Google Play store).

In addition, the mutation operators defined allowed us to assess the quality of the test cases generated by the iMPAcT tool. It was possible to identify some features to improve in the iMPAcT tool so that it can kill more mutants in the future and hence detect more failures. These features are: (A) improve the exploration algorithms so as to exercise more behavior; (B) deal with non-standard GUI controls; (C) extend the testing strategy to check, not only the background & foreground behavior but also situations in which the app is closed (with back button) and opened again. This would be helpful to test the correct use of persistent local storage; (D) extend the comparisons performed by the tool when checking if a test passes or fails, i.e., when checking if the state of the screen before going to background is equal to the state of the screen after coming back to foreground. For instance, compare colors and compare the widget of the apps on the home page.

V. Threats to validity

There are some threats to validity associated to the results we have presented. In terms of external validity, our results might not generalize to other apps. The overall study was performed over 56 apps. This set could be bigger, however, we were able to generate 611 mutants from this set of apps and analyze them automatically with the iMPAcT tool. It should also be noticed that the iMPAcT tool can take up to 20 minutes to run and analyze each mutant. Also, we analyzed the mutants manually in order to understand why the iMPAcT tool was not able to kill some mutants which also took time. To mitigate this threat we selected real-world apps of different categories.

In terms of internal validity, our experiment may not be completely free of errors because of 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 [19]. If the exploration does not reach the state change provoked by our mutation operator, the tool will not be able to kill the mutant, which does not mean that the mutant could not be killed by the testing approach implemented. To mitigate this threat we have used all different exploration methods supported by the iMPAcT tool [19]

In addition, we ran the iMPAcT tool over the original version of the apps and over the mutated versions of the apps but, since the testing strategies used by the tool have some degree of randomness, and besides using the same testing strategies to test different versions of an app, we cannot ensure that the events injected are exactly the same in both versions. In this particular case, if we had saved the events performed over the original app to replay them over the mutated versions, we could have stopped testing earlier than we wanted and killed more mutants. This might happen because the test could not be able to interact with a GUI element that was present in the original app in a certain point in time but not in the mutated version (Figures 1 and 3) or vice-versa (Figure 2). As a consequence, the test would fail (and would kill the mutant). So, to exclude this problem, we ran the iMPAcT tool over the original and over the mutated versions so as to just interact with the GUI elements present at each point in time. With this approach we are able to exclude problems such as trying to interact with GUI elements that do not exist but, eventually, kill less mutants; explore more behavior of the mutated versions of the apps; and kill more mutants related to the background & foreground behavior, which is the goal of this research. To mitigate this, we have ran different exploration methods provided by the iMPAcT tool over the original and the mutated versions of each app.

Still regarding internal validity, we have performed a manual evaluation of the background & foreground behavior of the apps which is subject to errors. However, this assessment was performed simultaneously by different team members to gain more confidence in the results. In addition, we also ran the iMPAcT Tool over the 50 original apps classified manually as correct. All of them were also marked by the iMPAcT tool as implementing the background & foreground behavior correctly.

VI. Related work

Software testing may be useful to find failures and to increase the confidence in the software correctness. However, mobile testing adds new challenges requiring different and specialized new testing techniques [2].

A. Testing mobile apps

There are different Mobile testing tools that can be classified according to different criteria [7]. Considering the testing activities they support, we can split them into, some, that can only automate the test case execution and, others, that are also able to automate the generation of the test cases.

Examples of tools for the former set are the script-based, in which 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), and Capture/Replay tools, in which the tester builds the test scripts by recording his actions interacting with the tool to replay such script later.

In the latter set, we can find random and model-based testing tools.

Random testing tools vary on the exploration algorithm used: it can be completely random or guided defining the next action to perform based on the knowledge acquired from the exploration performed until a specific moment (Active Learning).

Model-based testing tools vary on the model they use and the strategy defined to generate test cases from it. 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 [15], [16] and others have a predefined model which describes common traversal behavior to test overall mobile apps [4], [5], [20].

Despite all the classifications of the existing Mobile testing tools and guidelines to choose the best tool, a problem that remains is how to decide whether the set of test cases (manually or automatically built) is adequate to identify the failures someone aims to find. One way to assess the quality of test case is through mutation testing.

B. Mutation of mobile apps

Deng et al. [10] proposed the use of mutation testing in the context of Android apps. They designed and implemented eight mutation operators and developed a tool to create, install and execute mutants. The mutation operators were designed to exercise elements of the language that are specific to Android apps. They are divided in four classes: 1) intent operators; 2) event handler related; 3) activity lifecycle related; and 4) XML related. In this way, mutations are not applied only to programs (Java files in this case) but also to XML files, which in Android apps also determine behavior. In the empirical evaluation the authors conclude that using code coverage criteria would be as effective as using their operators thus there would be room to improve mutation operators for Android.

A problem pointed by Deng and colleagues is that there was no fault model for Android apps, so they had to design mutants based only on Android syntax. The work of Linares-Vsquez et al. [8] improves this aspect since they first defined a taxonomy of faults for Android apps and then designed 38 mutation operators, based on such taxonomy. The taxonomy identifies 14 classes of faults. Some are related to Android specific features such as Activities and Intents and GUI, others with Java specific features such as Collections and Strings and others with both Java and Android features such as I/O and Threading. The operators are also applied to program files and XML as well. They were implemented in a mutation tool name MDroid+. According to the authors, comparing the tool and the operators with traditional mutation tools, they found that their approach is more representative of Android faults and produces less useless (stillborn and trivial) mutants.

VII. Conclusions

This paper has presented 3 mutation operators created to mimic problems related to inadequate preservation of transient UI state when an app goes to background and returns back to foreground. It has presented a motivations example where it is possible to see situations from real applications where GUI controls disappear or appear or its state changes.

The mutation operators were defined following a process in which we had to study the lifecycle of Android apps and the different ways available to preserve transient UI state.

For each mutation operator defined, we presented the reason behind its implementation and performed some tests in order to check if they were able to mimic the problems we aimed to simulate.

This paper also presents an exploratory study. This study was performed over 56 real apps found on Google Play Store. The criteria to select these apps was presented. One of the goal was to build a set of apps diverse covering different categories.

We were able to generate 611 mutants. These mutants were used to evaluate the quality of the test cases generated by a mobile testing tool. In this experiment we have used the iMPAcT tool but any other mobile app could be used.

We have analyzed the results and explained the rational behind them. We also proposed some improvements to the iMPAcT tool so as to kill more mutants in the future. The threats to validity were also discussed.

In the future, we intend to explore other ways to save UI state and study the impact of mutation operators based on those methods in the background & foreground behavior. In particular, we aim to study if it is possible to generate mutants based on the behavior of the *onSaveInstanceState()* method of the *View* class (which allows to save, for example, the current cursor position in a *text view* and the currently selected item in a *list view*) and the *onSaveInstanceState()* that controls whether the saving of the related view's state is enabled.

In addition, we did not find any example within the 56 apps used in the case study using *ViewModel*. So, we aim to extend the study performed so far in order to build mutation operators based on the *ViewModel*.

We also aim to perform experiments to assess if the mutation operators defined also have impact on other specificities of mobile apps such as orientation change [25].

References

[1] Android platform: "ViewModel Overview", //developer.android.com /jetpack/arch/viewmodel, accessed in October 2018.

[2] 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.

[3] Android platform: "Saving UI State", //developer.android.com/topic /libraries/architecture/saving-states, accessed in October 2018.

[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] Android platform: "Intent", https://developer.android.com/reference/an droid/content/Intent, accessed in October 2018.

[7] 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.

[8] Mario Linares-Vsquez, Gabriele Bavota, Michele Tufano, Kevin Moran, Massimiliano Di Penta, Christopher Vendome, Carlos Bernal-Crdenas, and Denys Poshyvanyk. "Enabling mutation testing for Android apps". In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017). ACM, New York, NY, USA, 233-244, 2017.

[9] Eugene Spafford, "Extending mutation testing to find environmental bugs", Software: Practice and Experience, 20(2), 1990.

[10] Lin Deng and Jeff Offutt and Paul Ammann and Nariman Mirzaei, "Mutation operators for testing Android Apps", in Information & Software Technology, vol.81, pp.154–168, 2017.

[11] Gartner Inc., "Predicts 2017: Mobile Apps and Their Development", Adrian Leow, Van L. Baker, Richard Marshall, Magnus Revang, and Jason Wong, December 1, 2016.

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

[13] M. Linares-Vásquez, et al., "Enabling Mutation Testing for Android Apps", In Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 48, 2017.

[14] K. Moran, et al., "MDroid+: A Mutation Testing Framework for Android", in ICSE 18 Companion: 40th International Conference on Software Engineering, Gothenburg, Sweden, May 27–June 3, 2018.

[15] 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.

[16] 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.

[17] S. C. P. F. Fabbri, J. C. Maldonado, M. E. Delamaro, P. C. Masiero, "Proteum/FSM: A Tool to Support Finite State Machine Validation Based on Mutation Testing", Proceedings of the XIX SCCC – International Conference of the Chilean Computer Science Society, pp. 96-104, 1999.

[18] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero, M. E. Delamaro, "Mutation Analysis Applied to Validate Specifications Based on Petri Nets", Proceedings of the FORTE'95 – 8th IFIP Conference on Formal Descriptions Techniques for Distribute Systems and Communication Protocols, Montreal, Canada, pp. 220–229, 1994.

[19] Inês Coimbra Morgado and Ana C.R. Paiva, "Impact of Execution Modes on Finding Android Failures",in The 7th International Conference on Ambient Systems, Networks and Technologies (ANT 2016), in Procedia Computer Science", vol.83, pp. 284–291, 2016.

[20] Inês Coimbra Morgado and Ana C. R. Paiva, "Mobile GUI Testing", in Software Quality Journal, 2017.

[21] Inês Coimbra Morgado, Ana C. R. Paiva, "Test Patterns for Android Mobile Applications", Proceedings of the 20th European Conference on Pattern Languages of Programs (EuroPLoP 2015), Kloster Irsee in Bavaria, Germany, July, 2015.

[22] Android platform: "Understand the Activity Lifecycle", //developer.android.com/guide/components/activities/activity-lifecycle, accessed in October, 2018.

[23] 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 QUATIC2014 9th International Conference on the Quality of Information and Communications Technology (QUATIC), September 23rd, 2014.

[24] Inês Coimbra, Ana C. R. Paiva, João P. Faria, Rui Camacho, GUI Reverse Engineering With Machine Learning, in RAISE'12 - Workshop on Realizing Artificial Intelligence Synergies in Software Engineering, June 5, Zurich, Switzerland, 2012.

[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 Software Testing, Verification and Reliability Journal (STVR), 6 November 2017.

[26] Pedro Costa, Ana C. R. Paiva, Miguel Nabuco, "Pattern Based GUI Testing for Mobile Applications", in 9th International Conference on the Quality of Information and Communications Technology (QUATIC), 23–26 September 2014.