On the feasibility and challenges of synthesizing executable Espresso tests

Iván Arcuschin
FCEyN-UBA/ICC-CONICET
Argentina
iarcuschin@dc.uba.ar

Christian Ciccaroni
FCEyN-UBA
Argentina
cciccaroni@gmail.com

Juan Pablo Galeotti
FCEyN-UBA/ICC-CONICET
Argentina
jgaleotti@dc.uba.ar

José Miguel Rojas
University of Leicester
UK
j.rojas@leicester.ac.uk

ABSTRACT

Several tools have been proposed to automatically test Android applications, achieving outstanding results in terms of both code coverage and crash discovery. While useful for crash reproduction and bug-fixing, these tools usually do not present the generated interactions in a format that motivates developers to read and modify such tests later on. This hinders the ability of developers to add those tests to their existing test suites, or adapt them to new scenarios – common practices in modern software development where tests are maintained and evolve alongside production code.

In this work we present an empirical study on the challenges of automatically synthesizing Espresso test suites from sequences of interactions over widgets. We build on top of the MATE testing tool and implement a prototype that enables this study. The prototype is then evaluated on 12 open-source Android apps, followed by an analysis and discussion of challenges and limitations. We also include feedback from developers of open-source projects and an industrial app.

Our empirical study shows that the creation of Espresso tests is difficult, mostly due to the lack of unique properties to unambiguously identify specific widgets in the UI. This problem is aggravated in some cases by the incomplete or ambiguous definition of GUI components and layouts. It also points out that further research is needed to find ways to improve the testability of Android apps either manually or automatically. Nonetheless, the feedback received indicates that the synthesized Espresso tests are still useful for projects with few or no test cases, serving as a starting point for creating new ones.

KEYWORDS

Android test generation, Espresso tests, widget-based test generators

AST '22, May 17–18, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9286-0/22/05…$15.00
https://doi.org/10.1145/3524481.3527234

ACM Reference Format:
Iván Arcuschin, Christian Ciccaroni, Juan Pablo Galeotti, and José Miguel Rojas. 2022. On the feasibility and challenges of synthesizing executable Espresso tests. In *IEEE/ACM 3rd International Conference on Automation of Software Test (AST '22), May 17–18, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3524481.3527234

1 INTRODUCTION

As the use of mobile devices increases, the demand for mobile applications (known as *apps*) increases as well. Despite their growing popularity, *apps* tend to contain defects which can ultimately manifest as failures (or *crashes*) to end-users. Akin to other software domains, testing mobile apps allows developers to attain minimum quality thresholds for their applications. This process typically involves manual interaction with these apps under test and writing test cases for them. Developers write test cases not only to find faults but also to ensure that new features behave as expected and that changes made to the app do not break existing functionality (i.e., do not contain regressions). However effective, testing remains a time-consuming, error-prone, and costly task [33, 41].

Espresso [3] is a testing framework that provides an API to help developers write concise, reliable, and human-readable Android UI tests. Espresso is the only UI testing framework with substantial adoption amongst app developers [24], owing its popularity to the fact that it is part of the Android Software Development Kit (SDK) and that it provides mechanisms to prevent flakiness and to simplify the creation and maintenance of tests.

Multiple automated tools have been proposed in recent years for testing Android apps and improve their quality [21, 34, 44, 47, 48]. Nevertheless, just a handful of those tools target the Espresso format. What is worse, not all of these tools generate test cases in an executable and human-readable format. Most of them use the outdated Robotium [6] framework.

Therefore, many of those tools for automatically testing Android apps do not yield their test cases in a format that motivates developers to read and modify such tests later on. Thus, their adoption by developers who want to preserve valuable tests and re-run them periodically, e.g., in continuous integration, is hampered. Developers find themselves comfortable when writing test cases in the Espresso format [24]. We argue that generating test cases in Espresso would enable developers to preserve the generated tests in their test codebases, refactor them if needed, or reuse them as inspiration or starting points to augment their test suites.

In this work, we explore the feasibility of leveraging or extending existing tools as opposed to developing new ones from scratch. We present an empirical study on the challenges of automatically generating reproducible and human-readable Espresso test suites from sequences of interactions over widgets. For this purpose, we extend the MATE [27] testing tool to automatically translate sequences of widget actions into executable Espresso tests. This extension consists of two parts. First, we refactored MATE to output the description of the sequences of widget actions in JSON format. Second, we implemented an Espresso code synthesizer to translate this JSON document into readable and reproducible Espresso tests that match the original behavior as faithfully as possible.

The contributions of this article are:

- An extension to the MATE [27] widget-based testing tool to automatically synthesize Espresso tests.
- An empirical study focused on the *correctness* of the synthesized Espresso test cases. This includes a large quantitative and qualitative analysis to assert that the intended semantics of the original sequences are preserved.
- An analysis and discussion of the challenges and limitations of automatically generating Espresso tests from widget-based test generators. In order to understand, the *usefulness* of tools such as the one proposed for developers and the Android community, we also present the feedback collected from *pull requests* sent to open-source projects and from a questionnaire answered by developers of an industrial app.

Our empirical study yields the following insights:

- The creation of Espresso tests is difficult, mostly due to the lack of unique properties to unambiguously identify specific widgets in the UI. This problem is aggravated in some cases by the incomplete or ambiguous definition of GUI components and layouts. This points out that further research is needed to find ways to improve the testability of Android apps either manually or automatically.
- It is important that the test generation tool used to perform the exploration executes the actions on views that are reachable for Espresso. Furthermore, in order to alleviate the problem of identifying widgets in the UI, the information provided by the testing tool should be plentiful and accurate.
- Synthesized Espresso tests are useful for projects with few or no test cases and they can serve as a starting point for creating new test cases. They also provide a quick and simple way to increase the project's coverage. Nevertheless, these synthesized Espresso tests should include a clear description of the intent of each test case. Also, whenever possible, test cases should target a specific user scenario.

The remainder of this article is organized as follows: Section 2 overviews existing frameworks and tools for testing Android apps, with special emphasis on tools that generate Espresso tests. Section 3 presents the necessary background. Section 4 shows the implementation details of the prototype used for the empirical study. Section 5 presents the empirical study and discusses results. Section 6 presents an analysis and discussion of the challenges and limitations of synthesizing Espresso tests, as well as lessons learned during this work. Finally, Section 7 concludes the paper and outlines future work.

2 RELATED WORK

Several testing frameworks have been created to help Android developers write test cases and automate test execution. JUnit is a very popular framework for Java unit testing. In Android projects, pure JUnit test cases can be used only to exercise classes that have no interaction with the Android framework. In order to test the GUI components of an app (e.g., Activities, Widgets) an instrumentation-based framework is typically needed. These frameworks use Android Instrumentation [9] to inspect and interact with the Activities under test. Some well-known frameworks for Android GUI testing are Robotium [6], Appium [1], Calabash [2], MonkeyRunner [4], UIAutomator [7] and Espresso [3]. Of these frameworks, Robotium and Calabash are deprecated and no longer maintained. Notably, the Robolectric [5] framework allows unit testing of GUI components in a simulated Android environment inside a JVM, without the need of an emulator or device.

According to Cruz et al. [24], Espresso is the most used framework for GUI testing, with a steady increase in adoption in recent years. UIAutomator, Robotium, and Appium are used by very few projects, and AndroidViewClient, Calabash, Monkeyrunner, and PythonUIAutomator are not used at all. To understand if the Android testing tools also conform to these trends, we examined 101 papers from the Android testing literature and categorized each one by the type of output provided. The result was that only few tools target the Espresso framework as an output format. A great portion of them will not provide an executable output, e.g., yielding only crash reports. Most of the tools that do output executable test cases tend to either use a custom format (i.e., because they are using a custom engine to run them) or the Robotium framework. The list of the surveyed research papers can be found online [15].

Monkey [18], regarded as the state-of-practice, is a widely-used random-based testing tool for Android apps. It is provided with the Android SDK, hence its popularity among Android developers, but it only reports uncaught runtime exceptions during a random exploration. Moreover, Monkey does not allow users to replay sequences of events (i.e., test cases), which can be critical for understanding the cause of a crash. Similarly, other tools for automatically testing Android apps do not generate tests in a human-readable and reproducible format. For example, Sapienz [37] outputs a sequence of *atomic* actions intended to be used by machines (i.e., for re-executing failing cases), MATE [27] generates an accessibility report, while Dynodroid [36] and Stoat [43] only yield crash reports. Even recent tools such as Ape [30], Humanoid [35], TimeMachine [26], Q-testing [39] and ComboDroid [46] only yield coverage and crash reports.

Rohella et al. [40] briefly mention that their tool outputs Espresso tests, without providing any further details. COBWEB [32] uses Robolectric as an internal representation of tests and transforms them into Espresso tests at the end. They do not provide specifics on how this transformation is done. RacerDroid [45] modifies the Espresso framework to control event dispatching. It is unclear whether Espresso test cases generated by RacerDroid can be run outside the modified framework.

Espresso Test Recorder (ETR) [38] provides developers with a *record and replay* tool that eases the task of writing Android tests in Espresso format. However, ETR is not an automatic approach,

since the developer is in charge of providing the actual UI actions to be performed in the test. Hence, the use of ETR in practice is constrained by the amount of time and effort developers are willing to invest in interacting with the tool to record tests. Although ETR can listen for UI interactions that may be produced automatically (e.g., by a tool), it can only be started and commanded from the Android Studio IDE's UI. In other words, ETR can not be run as a standalone tool so integration with other tools for automated generation of Android inputs is restricted.

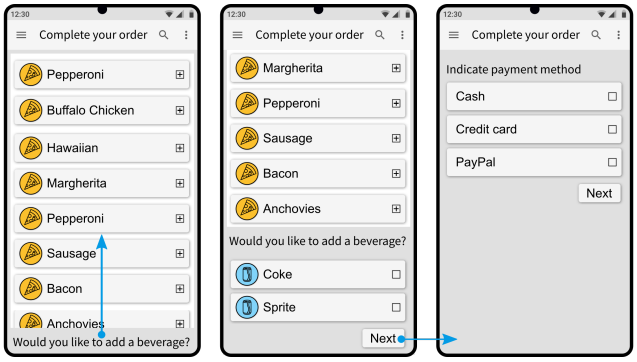
Barista [28] is a record and replay tool similar to ETR that provides better support for defining oracles and improved UI matchers in test cases. While their empirical study is mostly focused on comparing Barista against other record and replay tools such as ETR, ours is focused on assessing whether an existing automatic test generation tool can be extended to produce Espresso tests that preserve the original semantics achieved during exploration. This distinction is important since the sequences of interactions generated manually by developers for a record and replay tool might differ from the ones generated automatically by a tool. Nevertheless, some of the challenges presented in Barista's work [28] are similar to the ones presented in this article. Thus, we leverage the solutions and heuristics described by them to build our prototype.

DiffDroid [29] is a tool for automatically finding cross-platform inconsistencies in Android apps. It extends Monkey to generate sequences of interactions and encode them into a custom-defined trace, while simultaneously saving the UI hierarchies of the windows visited during exploration. Its implementation shows that it is also possible to extend automatic test generation tools that are not widget-based to synthesize Espresso tests. Still, DiffDroid's empirical study is focused on assessing if the tool detects cross-platform inconsistencies with a limited number of false positives. They do not study whether the synthesized Espresso tests are semantically equivalent or not to the original sequences produced by Monkey.

AppTestMigrator [20] is a tool for migrating test cases in Espresso format between apps of the same category (e.g., banking apps). It leverages commonalities between UIs to automatically migrate existing tests from one app to the other. Although AppTestMigrator's empirical study provides insights into how accurate the tool is in migrating tests, these test cases were written by developers and, as mentioned before, might differ from automatically generated ones.

Coppola et al. [22] present an approach for translating test scripts from visual-based tools into Espresso test cases. The motivation of their work presents similarities with ours but the approaches are different. For example, their implementation requires the app under test to be instrumented to complement the logs from the visual-based tool. Their experimental evaluation comprises 60 tests created manually by the authors (we use existing widget-based test generators as input) and does not address whether the semantics of the initial visual-based test scripts is preserved (cf. RQ 2). Overall, however, their work aligns with ours in recognizing the need to generate re-executable Espresso tests.

It is worth mentioning that some of the challenges and limitations presented in this article have also been mentioned in the related work [22, 23, 28, 29, 38]. Nevertheless, in most of these works the test cases are written manually by a human participant of the experiment or an outside developer (e.g., projects crawled from GitHub). In the few using automatically generated test cases,



(a) Motivating Android UI example. The user *swipes* up to reveal the rest of the list, and then *clicks* the "Next" button to reveal the second page in the form.

```
<CardView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <ImageView android:id="@+id/pizzaIcon"
        android:contentDescription="@string/pizza_icon" />
    <TextView android:id="@+id/pizzaName"
        android:text="@string/pepperoni" />
    <ImageButton android:id="@+id/addItem"
        android:contentDescription="@string/add_item_button" />
</CardView>
```

(b) XML example for "Pepperoni" row in Figure 1a.

Figure 1: Android UI example and associated XML code.

the challenges and limitations of synthesizing Espresso tests are not fully presented and analyzed. In contrast, this article provides an empirical study evaluating the feasibility and challenges of synthesizing Espresso tests from widget-based test cases automatically generated by a state-of-the-art-tool. We then perform a quantitative and qualitative analysis of the results. Such systematic analysis not only presents a detailed explanation of the challenges and limitations, but it also shows how pervasive each of these issues are.

3 BACKGROUND

3.1 Android UI

Android applications are composed of *Activities*. Each Activity defines a *window* with which the user can interact. An Activity consists of a UI and code to respond to UI-specific actions (e.g., clicking a button, typing in a text field, etc.) and/or system-wide events (e.g., low battery, lack of network connectivity, etc.).

The Activity's UI can be defined programmatically or via *XML* files (as shown in Figure 1b). In both cases, the developer must provide a *View Hierarchy* that the operating system will parse and use to render the screen. A View Hierarchy is a tree with a root *View* positioned at the top and child Views positioned as branches. A *View*, the basic building block for these hierarchies, occupies a rectangular area on the screen and is responsible for drawing and event handling. The View class is the base class for *Widgets*, which are used to create interactive UI components (e.g., buttons, text fields, etc.). The *ViewGroup* subclass is the base class for layouts, which are invisible containers that hold other Views (or other ViewGroups) and define their layout properties. Figure 1 shows an example of a UI and part of its hierarchy. A user interacts with a

```
[
  {
    "actionType": "CLICK",
    "parameters": "",
    "widget": {
      "classname": "android.widget.TextView",
      "resourceID": "com.myapp:id/button",
      ...widget attributes...
      "parent": {
        "classname": "android.widget.LinearLayout",
        "resourceID": "com.myapp:id/activity_layout"
      },
      "children": []
    },
  }
]
```

(a) Example of JSON action descriptor.

```
@Test
public void clickButtonTest() {
    onView(allOf(
            withId(R.id.button),
            withParent(withId(R.id.activity_layout))
        )).perform(click());
}
```

(b) Example of an automatically generated Espresso test.

Figure 2: Example of the JSON input (2a) and the generated Espresso test (2b).

mobile application by performing actions on the visible widgets. For example, clicking the "Add item" `ImageButton` in the rows of Figure 1a. Among others, some standard actions that can be performed on widgets are *click*, *long click* and *scrolling*.

3.2 The Espresso Testing Framework

Espresso [3] is a testing framework for writing Android UI tests. It was released in October 2013, and since its 2.0 release it is part of the Android Support Repository and officially supported by the Android ecosystem. This, coupled with the fact that it enables developers to write concise and reliable UI tests, has made the framework very popular among developers [24].

In a nutshell, Espresso provides an API that lets developers emulate user interactions with the app under test programmatically. This API is divided into three parts. *View Matchers*, to find views in the current view hierarchy (e.g., with a certain id, with a given caption, etc). *View Actions*, to perform actions on the views (e.g., clicking, long clicking, clearing a text, etc). *View Assertions*, to assert the state of a view (e.g., whether a view is displayed or not, whether it is above another view, etc). The framework also allows developers to select which activity should be run at the beginning of each test using a `@Rule` annotation. Furthermore, Espresso improves test case reliability by performing pending actions only when the application under test is idle.

Figure 2b shows an example of an Espresso test. The test selects a view with a "`button`" id and parent with "`activity_layout`" id. Then, it performs a click action on the matched view. If the view was not found, or several views match the selection criteria, the test execution will raise an exception.

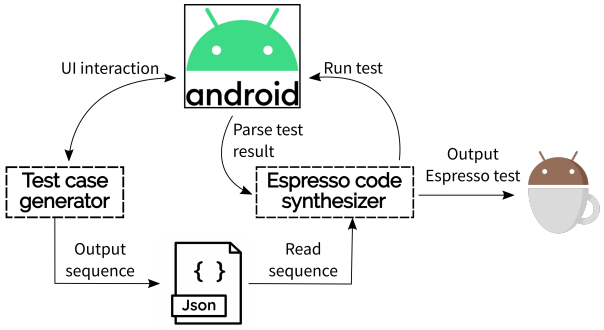


Figure 3: Architecture of the prototype.

3.3 Android Test Generation

Since the introduction of Monkey in the Android SDK, several tools have been proposed to automatically test Android mobile apps [21, 34, 44, 47, 48]. Dynodroid [36] aims to overcome Monkey's limitations by extending pure random testing with feedback-directed biases. Sapienz [37] applies search-based testing. It has been deployed at Facebook [8] where it is used to generate crash-reproducing test cases [19]. Stoat [43] applies model-based evolutionary testing, using dynamic exploration to build and evolve a probabilistic UI state-transition model from which to gather test cases. MATE [27] is a tool for generating sequences of interactions for Android, based on information from the UI. Although originally designed to find accessibility problems (e.g., a missing content description for a visible component), it has recently been extended to study different evolutionary algorithms for test generation [42].

The aforementioned tools internally represent the interactions with the mobile app under test using different formats. For example, Sapienz represents interactions as sequences of *atomic actions* (click, long click, etc.) over specific $\langle x, y \rangle$ screen coordinates. However, many of them use the UIAutomator [7] tool for finding available UI events and executing them, and thus they represent interactions as sequences of actions over widgets on the UI (called *widget actions*). Some of the tools in this group are: Stoat, Dynodroid, MATE, Ape [30], ComboDroid [46], TimeMachine [26] and Q-testing [39]. We refer to them as *widget-based* Android testing tools.

4 SYNTHESIZING ESPRESSO TESTS

The overall architecture of our prototype is depicted in Figure 3. We considered MATE, Dynodroid and Stoat to use as the underlying test case generator for the empirical study since all of them are *widget-based* state-of-the-art tools. We chose MATE [27] because it is the most modern of the three tools, it is in active development, we had previous experience using it, and, more importantly, because it had already been applied to a curated list of apps containing Espresso tests. We extended MATE in order to produce the JSON file shown in Figure 3. This document describes the sequences of widget actions that will constitute each Espresso test case. Figure 2a shows an example of JSON output. For each widget action, the type of action is collected, as well as any extra parameter needed (e.g., the text when typing in a text field). Each widget action also points

Algorithm 1: Translating sequences of widget actions into an Espresso test suite

```
   Input  : JSON file f
   Output : Test suite S
1  Suite_E ← {}
2  WActionSeqs ← ParseWidgetActionSequences(f)
3  for WActionSeq ∈ WActionSeqs do
       // Translate a sequence of widget actions
4      Test_E ← []
5      for WAction ∈ WActionSeq do
           // Translate a single widget action
6          TestCode ← BuildViewMatcher(WAction)
7          TestCode.append(BuildViewAction(WAction))
8          Test_E.append(TestCode)
9      if Test_E is not empty do
10         Suite_E ⟵ Suite_E ∪ {Test_E}
11 return Suite_E
```

to the targeted widget, its parents, children and specific attributes if there were any (e.g., displayed text). A full description of the JSON schema (including the action types provided by MATE) can be found in a public GitHub repository [10].

Algorithm 1 outlines how the *"Espresso code synthesizer"* module translates sequences of widget actions into an Espresso test suite. First, the `JSON` is parsed to obtain the sequences of widget actions (line 2). Then, an Espresso test suite is built by translating each sequence of widget actions into a (ideally) semantic-preserving Espresso call to the method `perform`. That is, for each widget action (e.g., the one shown in Figure 2a), the code synthesizer produces the corresponding Espresso statement to trigger the same action using the Espresso API. This is achieved by consecutively synthesizing the code for the appropriate *View Matcher* to select the widget and the corresponding `perform` operation on the selected widget (lines 6-8). For example, if the widget action is a *click*, then the appropriate perform operation would be `perform(click())`. Figure 2b shows an example where the `onView()` invocation selects the targeted widget on which `perform(click())` is executed. Notice that the algorithm does not append any *View Assertion*. The implementation does not add such assertions since they are not strictly necessary to reproduce the behavior of the sequence of widget actions. Finally, the resulting Espresso test suite is written as a single Java file to the test directory specified by the developer.

We faced several challenges during the implementation of the prototype. To mitigate them, we relied on the heuristics proposed by the official Espresso documentation and the state-of-the-art literature. Specifically, we used the heuristics proposed in related work to mitigate the problem of *widget disambiguation* (i.e. [28, 29, 38]). This problem arises because Espresso's `onView()` method, used to select the target widget of an action, takes a *View Matcher* that is expected to match a unique widget within the current view hierarchy. Often, the desired widget has a unique resource identifier which can be used to unequivocally target it (i.e., using the Espresso `withId` matcher). However, there are many legitimate cases in which the target widget may not have a resource identifier or the identifier may not be unique. In the latter scenario, an attempt to use the `withId` matcher will yield an `AmbiguousViewMatcherException`. As an example, some of the heuristics applied were: using the parent's resource id (as shown in Figure 2b), using the children's resource id, and using the view's text (or content description) if non-empty.

Table 1: Subjects used in empirical study. Last activity was inspected at 2020-06-10.

Subject	Last activity	#Android Activities	#Tests in project
PoetAssistant	2019-10-16	7	152
Equate	2020-05-31	2	6
OneTimePad	2017-11-01	2	0
Orgzly	2020-05-12	13	460
MicroPinner	2019-08-31	1	23
OCReader	2020-06-09	2	17
HomeAssistant	2018-02-07	3	3
OmniNotes	2020-06-05	5	71
Kontalk	2020-04-21	14	59
KolabNotes	2020-03-20	9	0
ShoppingList	2019-03-19	8	7
MyExpenses	2020-06-06	13	521

5 EMPIRICAL STUDY

Ideally, all synthesized Espresso tests would preserve the intended semantics of the original widget-action sequences from which they were generated. However, as discussed in Section 4, this may not be always possible. The following research questions aim to reveal how pervasive this problem is. A replication package of this study can be found online [17].

RQ 1: Do the synthesized Espresso tests reliably replicate the coverage achieved by widget-action sequences?

RQ 2: Do the synthesized Espresso tests reliably replicate the UI states achieved by widget-action sequences?

RQ 3: Which are the most common causes for failure in the synthesized Espresso tests?

5.1 Experimental setup

5.1.1 Selection of Subjects Under Test. We used as experimental subjects the 12 open-source apps with Espresso setup collected and used by Eler et al. [27]. These apps are adequate as subjects for our research questions since MATE can handle them, and most of them have recent activity. They are shown in Table 1.

5.1.2 Subjects configuration. We configured each subject accordingly since most of them had several build variants (e.g., release, debug, etc.) and product flavors defined (i.e., specifying different features). The *"Espresso code synthesizer"* module automatically detects the Espresso version configured in the subject (e.g., using AndroidX or Android Support libraries), which prevents dependency problems during execution.

5.1.3 Implementation. We made minor changes to MATE's code in order to use its output. In particular, we added a small code fragment (55 LOC using the Jackson [13] library for handling JSON documents) that dumps the Java objects representing the final population to a JSON file. We used MATE's *Random Exploration* algorithm for the generation of widget-based test cases. Random exploration is a fairly simple algorithm which has been used in MATE for Android test case generation before [42]. The algorithm works by continuously sampling the search space until it runs out of time budget. On

each iteration, a completely new individual (i.e., test case) is created. If this new individual increases the overall coverage achieved up to that point, the individual is added to the final population. Therefore, the final population consists of all the individuals that increased the overall coverage during exploration. It is worth pointing out that the experiments were fully automated and no manual intervention was provided (e.g., logins) during MATE's exploration.

5.1.4 Experiment Procedure. The experiments were run on a PC with Ubuntu 18.04. The CPU was an Intel® Core™ i7-7700 @ 3.60GHz × 8 cores and the RAM was 32 Gb. We used Android Pie emulators (API 28). To account for the randomness of the chosen algorithm and mitigate non-determinism, we executed each experiment 5 times on each subject. We set a maximum time budget of 1 hour for each execution of MATE. We conservatively doubled the original 30 minutes time budget used by Eler et al. [27] to mitigate any emulator or hardware difference. No time budget was set for the *"Espresso code synthesizer"* module since the translation per se (lines 5-8 of Algorithm 1) takes only a few seconds (negligible compared to exploration time).

5.1.5 Experiment Analysis. We obtained statement coverage for each executed test cases using JaCoCo [14]. We manually inspected that the coverage achieved by MATE's exploration and the corresponding Espresso tests was being correctly reported by JaCoCo. We excluded from the JaCoCo analysis some of the classes that might be in a subject's APK but were not directly written by their developers (e.g., Android libraries, auto-generated classes, etc).

For each Espresso test generated, we also collected the number of calls to the *perform* method that failed. Namely, how many actions in the original test cases were not properly translated to Espresso.

Furthermore, to understand if the generated test cases preserve the same semantics as the original widget-based test sequences, we collected screenshots during its execution. In particular, we took a screenshot before each action and a single final screenshot at the end of the test. By doing this, we aimed at recording any visible difference between test executions. For comparing the screenshots, we performed a pixel-by-pixel comparison of each image using the *compare* tool from the *imagemagick* [11] open-source software suite. The comparison was done with a 20% fuzz factor [12] that helps to ignore minor differences between the two images. Then, we use the "AE" (short for "Absolute Error") special metric to count the actual number of pixels that were masked, at the current fuzz factor. We deem two screenshots to be *diverging* if the proportion of pixels that differ is greater than 5%. In that case, the action previous to the screenshot is a point in the test execution were the original widget-based test sequence and the synthesized Espresso test diverged. Both the threshold and fuzz factor chosen were manually validated using several examples of screenshots taken from the subjects in the experiment. We also provide further validation in Section 5.2.2.

5.2 Results

5.2.1 RQ 1: Do the synthesized Espresso tests reliably replicate the coverage achieved by widget-action sequences? Figure 4 shows for each subject the overall coverage achieved by MATE and the synthesized Espresso test cases. The latter achieve a similar coverage

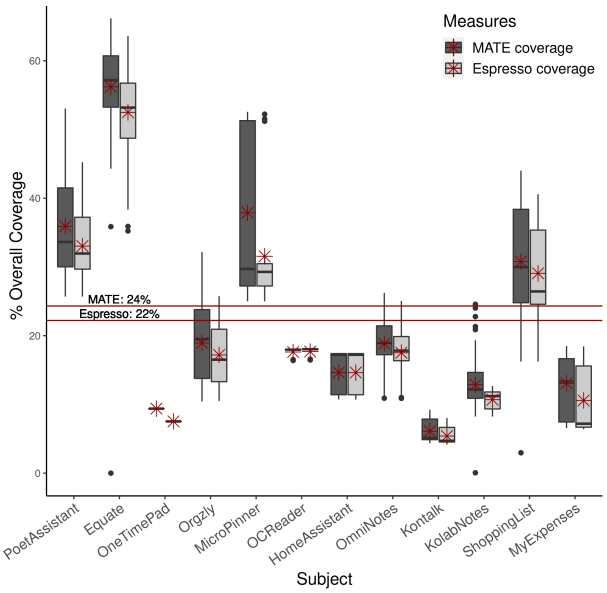


Figure 4: Overall coverage achieved for each subject. Middle line of each boxplot marks the median, black circles represent outliers, ⋆ symbol shows the mean, and the red line represents the mean of all coverages (22% for the Espresso tests and 24% for MATE).

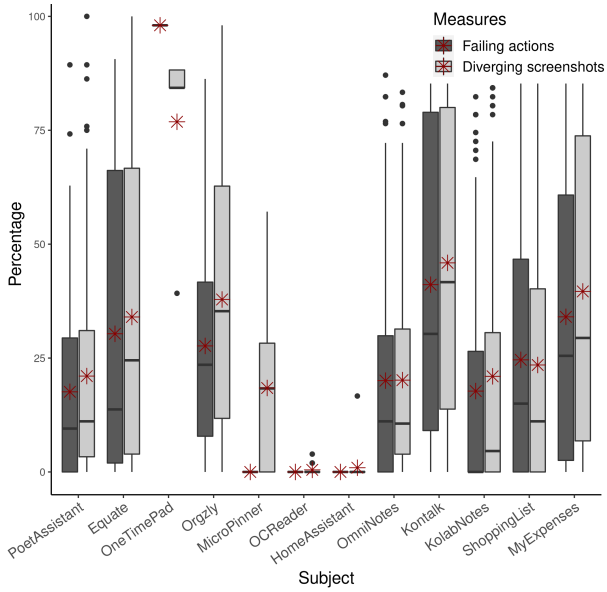


Figure 5: Overall failing actions and diverging screenshots found while running the Espresso test cases for each subject. Middle line of each boxplot marks the median, black circles represent outliers, ⋆ symbol shows the mean.

to MATE's widget-action sequences. Although the differences between coverage are small, in most cases the synthesized Espresso test cases achieved lower coverage than MATE. We conjecture that this behavior (which is consistent across many subjects) is caused by the fact that there is at least one failing action on each test suite when executed. Figure 5 shows the percentage of actions that failed

Table 2: Number of screenshots that were correctly or incorrectly marked as *diverging* screenshots by the pixel-by-pixel comparison.

Marked status	Correctly	Incorrectly	**Total**
Diverging	1257 (36.97%)	58 (1.71%)	**1315 (38.68%)**
Same	1618 (47.59%)	467 (13.74%)	**2085 (61.32%)**
Total	**2875 (84.56%)**	**525 (15.44%)**	3400

when running the synthesized Espresso tests. If we look across subjects, we find that in total 30.19% of actions failed. Observing each test case, we find that on average 25.25% of actions failed.

RQ 1*: The synthesized Espresso tests preserve over* 90% *of the widget-based exploration coverage (*22% *vs.* 24%*).*

5.2.2 RQ 2: Do the synthesized Espresso tests reliably replicate the UI states achieved by widget-action sequences? Figure 5 also shows the percentage of *diverging* screenshots found while running the synthesized Espresso tests. Given the high number of diverging screenshots found (33.64% diverging screenshots across subjects and 29.18% per test case on average), we decided to perform a more detailed *qualitative* analysis of these screenshots, to better understand if the pixel-by-pixel comparison was unintendedly introducing errors in our results. In order to do so, we manually inspected more than half of the screenshots recorded by the first repetition of our experiment (**3400** pairs of screenshots, precisely). Each pair contains a screenshot taken during MATE execution of the widget-based sequence and a screenshot taken during the execution of its corresponding Espresso action.

Table 2 shows how many of the screenshots manually inspected were actually diverging or not. The pixel-by-pixel comparison was able to correctly detect the diverging status of 84.56% of the screenshots analyzed. Of the remaining 15.44%, 58 (1.71%) screenshots were incorrectly marked as diverging and 467 (13.74%) were incorrectly marked as the same screen. This shows that, although not perfect, the pixel-by-pixel comparison (along with the chosen fuzz factor and diverging threshold) is able to correctly determine if two UI states are the same or not in a significant number of cases.

In Table 3 we summarize the causes observed for the diverging screenshots. The main cause observed is "Wrong UI State", meaning all screenshots that are different as a consequence of a previous divergence in the test. For example, a failing *back action* at the beginning of a test case means that the following actions in the test will start from a different UI state than what was intended in the original sequence. This may indicate certain *fragility* in GUI tests, which is a known problem that has been studied in the literature [23]. The next cause observed is "Off Timing" which comprises all the divergences caused by small differences in timing when executing the test cases, which led to some actions failing.

"MATE Exited" refers to a particular bug in MATE's implementation that caused MATE to continue the exploration even after having exited the subject under test. This faulty behavior produced repeated screenshots of the emulator's *home screen* and other unrelated apps at the end of some executions.

Table 3: Details of causes for *diverging* screenshots.

Cause	Count (%)
Wrong UI State	1246 (72.27%)
Off Timing	216 (12.53%)
MATE Exited	129 (7.48%)
Failed Action	93 (5.39%)
Flaky Test	40 (2.32%)
TOTAL	**1724**

"Failed Action" represents all the screenshots that diverged because of an action failing prior to taking the screenshot. Lastly, "Flaky Test" refers to all the screenshots that diverged due to the test cases generated by MATE being flaky. For instance, the execution of an action that is date dependant (e.g., setting an alarm) produces a flaky test. As another example, some subjects retrieve information from Web services and the result changes over time (e.g., currency exchange rates). It is worth noticing that in this case the MATE sequences themselves can not be replicated, since the flakiness is caused by the behavior of the subject under test.

RQ 2*: The synthesized Espresso tests achieve similar UI states in about two-thirds of the cases. In the ones that differ, it is usually caused by a failing action at the beginning of the test.*

5.2.3 RQ 3: Which are the most common causes for failure in the synthesized Espresso tests? Given the high amount of missmatching pairs of screenshots that were caused by a previous *divergence* in the test, we decided to also study the first failing action of those tests. In total, we manually inspected 94 tests (the same ones considered in our manual analysis of screenshots). Of those 94, 31 did not contain any failing action. Table 4 shows the causes of the first failing action for the remaining 63 tests. We group the causes by which component in the whole architecture was responsible (i.e., which one should be changed to avoid the problem).

The *"Espresso code synthesizer"* module is responsible for 38.10% of the failing actions analyzed. Among the reasons for this we can enumerate the following. "JSON Schema" refers to all the cases in which a failing action could be fixed by enhancing the JSON schema with additional information that is already available in MATE. "Swipe Difference" refers to the failing actions that occurred by small differences in screen placement between the synthesized Espresso tests and the original widget-based test sequences. These changes led to some actions not being available or present in the screen at the moment of re-execution. "Ambiguous Matcher" represents the actions that failed due to any of the heuristics in the *"Espresso code synthesizer"* module for building the *View Matcher*.

The Espresso framework is responsible for 30.16% of the failing actions analyzed. "View Not Displayed" refers to any action that failed due to the target view not being displayed in the screen, which is a common Espresso limitation. "Framework Limitation" represents other known limitations of the Espresso framework, for example, when working with custom views. "Subject Not Supported" represents the actions that failed due to a specific implementation detail in the subject.

Table 4: Analysis of causes and components for failing actions

Component	Cause	Count (%)	Summary
Espresso code synthesizer	JSON Schema	16 (25.40%)	24 (38.10%)
	Swipe Difference	5 (7.94%)	
	Ambiguous Matcher	3 (4.76%)	
Espresso framework	View Not Displayed	12 (19.05%)	19 (30.16%)
	Framework Limitation	6 (9.52%)	
	Subject Not Supported	1 (1.59%)	
MATE	Wrong Info	9 (14.29%)	10 (15.87%)
	Accessibility Service	1 (1.59%)	
Subject	Flakiness	10 (15.87%)	10 (15.87%)
TOTAL		**63**	

After further inspection, we found that the errors in the category "View Not Displayed" were being worsened by MATE's behavior. Specifically, MATE sometimes performs actions on views that are not *displayed* on the screen (i.e., views not directly visible to the user). A typical instance of this is a screen with a long form on it: the views at the bottom are "visible", but not on *display*. The first screen in Figure 1a exemplifies this case: the beverage options and "Next" button are visible in the UI but not currently displayed to the user. This behavior of MATE collides with the way that the default Espresso actions work, which assumes that target views are always *displayed* to the user.

MATE is responsible for 15.87% of the failing actions analyzed. "Wrong Info" refers to actions that failed due to MATE providing erroneous information in the JSON. For instance, actions performed by MATE on UI elements *outside* the app (e.g., in the top bar) break compilation of the corresponding Espresso tests, since the resource IDs of those views are not defined within the target app.

"Accessibility Service" represents the known limitations of the Accessibility Service used by MATE to gather information in the screen. Some of these limitations led to incongruencies during the generation of Espresso tests: imprecise *class name* of views, incorrectly reporting the *hint* of a field as *text* input, missing *content description* properties, and providing texts with the wrong casing.

Finally, general flakiness in some subjects sums up to 15.87% of the failing actions.

RQ 3: *The most common causes for failure are: missing information in the JSON Schema, performing actions on widgets not displayed, erroneous information from MATE and overall flakiness in subjects.*

5.3 Threats to Validity

Threats to internal validity might result from how the empirical study was carried out. For the selection of the algorithm, we considered the ones studied in [42]. We decided to use the best performing algorithm in that study, i.e., Random Exploration. Since the algorithm Random Exploration is affected by non-determinism, we ran 5 repetitions on each subject with different random seeds. Also, parameter tuning can affect the performance of algorithms, so we used the same default values for all parameters across experimentation. These values were chosen based on [42].

Threats to external validity come from the fact that we only used 12 open-source subjects as case studies. To avoid selection bias for the open-source subjects, we explicitly decided to include *only* those apps that had been previously used in the empirical study of MATE [27], and that had already been set up to run Espresso tests. Nevertheless, it is important to note that another selection of subjects might result in different conclusions.

6 ANALYSIS AND DISCUSSION

In this section, we discuss the main challenges and limitations of synthesizing Espresso tests and present the insights learned from this study. We focus on technical challenges first and then analyze the existing barriers that might prevent developers from adopting a tool such as the one implemented in this article into their daily workflow.

6.1 Challenges & Limitations

Results from the *RQ 2* (Section 5.2.2) show that divergent UI states between a widget-based action sequence and its corresponding Espresso test are mostly caused by failing actions. The rest of the causes had a lower impact during the study: timing issues, flaky subjects and problems in the test generation tool.

Results from the *RQ 3* (Section 5.2.3) show the causes observed for failing actions. *The main challenge detected is the difficulty to properly indicate to the Espresso framework the view on which it should perform an action.* To do so, an Espresso test case generator (or developer) has to correctly build a corresponding *View Matcher*. Since the Espresso framework requires an action to be performed on a unequivocally identified view, this *View Matcher* must be built with great precision. Failure to do so causes the Espresso framework to raise an exception.

View Matchers can be built using resource IDs, which are not always unique, and many other view properties (e.g., text, children, parents, etc.) that may help narrow down the search. Therefore, *it is of the utmost importance that the information provided by the underlying test generation tool is plentiful and accurate.* Any missing or wrong information has a high chance of causing a failing action.

We also found that *it is important that the test generation tool performing the exploration executes the actions on views that are reachable for Espresso.* For instance, actions performed on views not *displayed* to the user will not work with Espresso default actions, and those performed on views *outside* the application will not work at all. Although there are some workarounds for the former (e.g., implementing custom actions), these do not work for all cases.

Lastly, as mentioned before, some issues are independent of the method chosen for synthesizing Espresso tests. Typically, subject flakiness may lead to sequences of widget actions that are not reproducible. Also, limitations of the Espresso framework prevent it from working with custom views without additional efforts.

In summary, the creation of Espresso tests is difficult due to the lack of unique properties to unambiguously identify the elements in the layouts. This problem is aggravated in some cases by the incomplete or ambiguous definition of GUI components and layouts. In this regard, further research is needed to find ways to improve

the testability of Android apps either manually or automatically. We outline below the particular technical limitations faced during the empirical study that may also be of interest to other researchers.

6.1.1 Widget disambiguation limitations. Despite the strategies implemented to mitigate the problem of widget ambiguity, some cases might not be possible to disambiguate:

Images with identical hierarchy but different content. This can happen, for example, if two different images are side by side. In this case, it is not possible to build a *View Matcher* that distinguishes both by the content (nor does the Accessibility Service used by MATE provide such info).

Widgets with identical hierarchy and content, but different siblings in the hierarchy. For example, the second screen in Figure 1a shows a short list of beverage options in which each row has the same checkbox, but all rows have different text. We might want to target a specific checkbox (e.g., the one in the "Coke" row) but if we only look at the parent and children hierarchy, as proposed in our JSON schema, there is no way to distinguish one from the other. This might be solved if we add siblings to the JSON schema, or if we provide the whole UI hierarchy tree in the JSON.

Widgets with identical hierarchy, content and siblings. For example, if we have duplicated rows in a list (identical including content, as shown for the "Pepperoni" row in the first screen of Figure 1a), then the only way to disambiguate them is by using their position on the list, but this information is not provided by the Accessibility Service, and is therefore not listed in MATE's output.

6.1.2 Lack of testability transformations. The concept of *testability transformation* [31] refers to any modification applied to a program to make the testing process more effective. In that sense, the process of automatically synthesizing Espresso tests may benefit from custom-made testability transformations. These transformations may help for example with the problem mentioned in the previous subsection by assuring that all widgets have a unique resource id. An instance of such transformation is presented by Coppola et al. [22], but further analysis is needed to measure its effectiveness.

Another scenario in which testability transformations may be useful is when an app blocks the *main thread* (i.e., the UI *thread*). Since the actions executed by Espresso are also run on the *main thread*, any blockage there prevents Espresso from working. Particularly, this happened in our study with the subject `OneTimePad`. Although this is a limitation of Espresso, we believe that the behavior implemented by `OneTimePad` is not standard and should not impact the broader range of apps. It is interesting to note that MATE does work for `OneTimePad`, since it runs as a standalone process in the device, and thus in a different *thread*.

6.1.3 Engineering limitations vs. Research challenges. So far, we have presented all challenges and limitations together. Nonetheless, it is useful to also separate these issues between incidental and fundamental ones. Incidental limitations are the ones that can be rewritten as an engineering problem. In other words, they are limitations faced as byproducts of the chosen implementation for the prototype or experimental setup. We identify the following:

- Ensuring that information gathered during test generation is correct, so *View Matchers* are built correctly afterwards.
- Providing all the information available during test generation to the *Espresso code synthesizer*.
- Finding ways to gather information on custom views, images and lists. This issue would also need the implementation of custom *View Matchers*.
- Minimizing errors due to swipe differences and timing issues to avoid overall flakiness.

As an example, the first item mentioned above could be eased by improving MATE in a way that it does not perform actions outside the AUT, and that the ones inside the AUT are only on views reachable for Espresso. Also, replacing the Accessibility Service used in MATE for another source of UI information would be really helpful. Fundamental challenges can be rewritten as a research problem. Namely, special algorithms or techniques need to be devised to find ways to solve or mitigate them. We deem the following as such issues:

- Insufficient Android information to unambiguously identify the elements in the UI layouts.
- Subject flakiness due to UI states that depend of external factors (e.g., time or Web services).
- Furthermore, lack of *testability transformations* that can be applied to the AUT to mitigate both issues aforementioned.

6.2 Adoption & Usefulness

To understand the usefulness and limitations of the synthesized Espresso tests for developers, we evaluated the effectiveness of the implemented prototype on the previously studied 12 open-source apps plus one industrial app. We submitted synthesized Espresso tests as *Pull Requests* for the open-source apps and hereby report on their acceptance rates. Finally, we provide the prototype to an industrial partner for its evaluation. We conducted a written interview with two developers of the Android development team. We asked these developers to outline the positive and negative aspects of the prototype, as well as feedback to improve the usefulness of the generated tests.

6.2.1 Open-source projects. We submitted a *Pull Request* with a single test case to each project public's code repository (e.g., in GitHub). To avoid any possible bias, we followed a strict selection process to decide which test to submit.

Firstly, we start by considering only those Espresso tests synthesized in the first repetition of our empirical study. Secondly, if the synthesized Espresso test case with the highest coverage for a project increased the project's overall coverage by 5% or more, it was selected for submission. When no test case was able to meet this requirement, we checked if there were Espresso tests that covered activities missing in the coverage of the existing project's test suite. If that was the case, we selected the one that brought the highest coverage increase to the project. If no test case brought more than a 5% coverage increase nor covered any new activity, we decided not to send a *pull request*.

Each pull request contained the selected test case, two auxiliary documented Java files, external dependencies if necessary, and a small explanation highlighting the contributions (i.e., the overall coverage increase and the new screens covered). For the subjects with explicit *contributing guidelines*, we manually reviewed the code submitted to make sure it adhered to them.

Overall, we selected 8 out of 12 subjects for sending *Pull Requests*. Of the 8 *Pull Requests* submitted, we received responses with comments for half of them (`ShoppingList`, `MicroPinner`, `KolabNotes` and `OmniNotes`). `KolabNotes` and `OmniNotes` merged the *Pull Request* into their test codebase and thanked the contribution. The maintainer of `OmniNotes` even said that *"Code coverage is always the best way to contribute!"*. The *Pull Request* submitted to `HomeAssistant` was closed without comments, and shortly afterwards the project was closed and *archived* by the maintainer. Our hypothesis is that, in this early stage, *the prototype's output is mostly useful for active projects without test cases or with some test cases but low coverage.*

6.2.2 Industrial project. In order to evaluate the usefulness of the prototype and its output, we reached out to an industrial partner. This industrial partner has a mobile application developed for handling credit card payments. The application has more than 30 different screens and more than 100k lines of code, using both Java and Kotlin as programming languages.

In terms of their testing codebase, the project has only unit tests (about a hundred). Although they do not measure coverage, they suspect it to be rather low. The project does not have integration nor system-level tests, nor do they use the Espresso framework.

We were able to show the prototype to two Android developers in the company. We asked them to try it out and answer a series of questions afterwards. The evaluation was performed remotely and asynchronous, due to the ongoing pandemic. The questions were sent via email. In their evaluation they limited the access to 25 out of 30 screens. This limit was imposed because the remaining screens interact with Bluetooth devices that are not available when using an emulator.

After trying out the prototype and inspecting the synthesized Espresso tests, they reported the following positive aspects. First, *for projects with none or few system-level tests it provides a quick and simple way to increase their coverage.* In other words, the generated tests were successfully executed and *they also serve as a starting point for creating new test cases.* Secondly, although there is no guarantee, Random Exploration allows the tool to find obscure bugs that otherwise might reach end users.

They also highlighted the following disadvantages. First, *the synthesized Espresso tests do not provide a clear description of what are the goals of each test case.* Secondly, due to the use of MATE's Random Exploration, test cases tend to become large (i.e., up to 50 actions). Such large tests become very fragile when one has a large app that is constantly growing. *Whenever possible, test cases should target a specific user scenario.* Also, Random Exploration might cause an uneven examination of the app. Likewise, these randomly generated tests have some undesirably redundant actions that could be removed. Thirdly, tests generated in one device configuration, need to be executed in the same configuration. This is caused by Android UIs changing their appearance depending on the size, density and orientation of the display.

Once the questionnaire was completed, we asked them whether they would consider incorporating the tests generated to their codebase. They answered that first they would need to have a clear description for each test case. But if that was provided, they said that they would add all reasonable tests without making any changes to them. They also commented that by adding those tests, the coverage of the project would increase. It is worth mentioning here that the problem of tests summarization is an open research challenge, and is not constrained to Android test generation [25].

Finally, we asked them what changes to the current prototype they would suggest to improve its *usability* (i.e., ease of use by developers), *adoption* (i.e., adding the tool into the workflow) and *integration* (i.e., the regular use of the tool). In terms of *usability*, they proposed: improving failure messages in test cases, adding test case descriptions, and adding the possibility to guide the exploration in a semi-automatic form (i.e., having a human-in-the-loop).In terms of *adoption* and *integration*, they mentioned that it is very important to provide integration plugins for the tools already used in a daily basis, e.g., Android Studio IDE. Also, in case it is not possible to fix the problem of having to run the tests in the same device they were generated, it would be good if the prototype could explore test cases in several devices at the same time.

7 CONCLUSIONS AND FURTHER WORK

We conducted an empirical study evaluating the feasibility and challenges of automatically synthesizing Espresso UI tests from widget-based action sequences. The study was carried out on several open-source apps and one industrial project. To further understand the challenges and limitations of automatically creating Espresso tests, the study was followed by a quantitative and qualitative analysis of the results.

Our prototypical implementation and experimental results suggest that leveraging existing testing tools to generate executable Espresso tests is a promising research direction. On the other hand, our study unveils concrete challenges that would need attention before a fully-fledged, industrial-strength solution can be devised.

This work contributes towards bridging the gap between existing Android app testing tools and developers requiring Android tests in Espresso format. We believe that the insights learned in this work are useful for the Android research community and applicable to other test generation tools that may want to translate their output to Espresso test cases. The prototype used for the study is open-source and can be found on GitHub [16].

As future work, we plan to explore different testability transformations that might improve the reliability of the synthesized Espresso tests. Besides, it would be interesting to extend the Stoat and Dynodroid testing tools to perform a similar study using sequences of widget actions that may differ from the ones generated by MATE. Another possible line of future work is to perform an empirical study using other testing frameworks such as Appium [1], in order to examine trade-offs between competing frameworks. Finally, we plan to assess and improve the quality of the synthesized Espresso tests in terms of fault-detection and non-functional properties, e.g., readability.

ACKNOWLEDGMENTS

This work is partially funded by UBACYT 2020 Mod 1 20020190100233BA, ANPCYT PICT 2019-01793 and H2020-MSCA-RISE-2018 BehAPI Project (grant agreement No. 778233).

REFERENCES

[1] [n. d.]. Appium: Mobile App Automation Made Awesome. http://appium.io/. (Accessed on 08/03/2020).

[2] [n. d.]. calabash/calabash-android: Automated Functional testing for Android using cucumber. https://github.com/calabash/calabash-android. (Accessed on 08/03/2020).

[3] [n. d.]. Espresso - Android Developers. https://developer.android.com/training/testing/espresso. (Accessed on 08/03/2020).

[4] [n. d.]. monkeyrunner – Android Developers. https://developer.android.com/studio/test/monkeyrunner. (Accessed on 08/03/2020).

[5] [n. d.]. Robolectric. http://robolectric.org/. (Accessed on 08/03/2020).

[6] [n. d.]. RobotiumTech/robotium: Android UI Testing. https://github.com/RobotiumTech/robotium. (Accessed on 08/03/2020).

[7] [n. d.]. UI Automator – Android Developers. https://developer.android.com/training/testing/ui-automator. (Accessed on 08/03/2020).

[8] 2018. F8 2018: Friction-Free Fault-Finding with Sapienz. https://developers.facebook.com/videos/f8-2018/friction-free-fault-finding-with-sapienz/.

[9] 2022. Android Instrumentation. https://developer.android.com/reference/android/app/Instrumentation.html.

[10] 2022. Full description of the JSON schema. https://github.com/FlyingPumba/etg/blob/master/schema.json.

[11] 2022. ImageMagick's compare program. https://imagemagick.org/Usage/compare/.

[12] 2022. ImageMagick's Fuzz Factor. https://legacy.imagemagick.org/Usage/color_basics/#fuzz.

[13] 2022. Jackson JSON library. https://github.com/FasterXML/jackson.

[14] 2022. JaCoCo coverage tool. https://www.eclemma.org/jacoco.

[15] 2022. Online related work. https://github.com/FlyingPumba/etg-paper-replication-package/blob/master/related-work.pdf.

[16] 2022. Prototype's code. https://github.com/FlyingPumba/etg.

[17] 2022. Replication package. https://github.com/FlyingPumba/etg-paper-replication-package.

[18] 2022. UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey.html.

[19] Nadia Alshahwan, Xinbo Gao, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, Taijin Tei, and Ilya Zorin. 2018. Deploying Search Based Software Engineering with Sapienz at Facebook. In *SSBSE (LNCS)*. Springer.

[20] Farnaz Behrang and Alessandro Orso. 2019. Test Migration Between Mobile Apps with Similar Functionality. In *ASE*. IEEE, 54–65.

[21] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet? (E). In *ASE*. IEEE Computer Society, 429–440.

[22] Riccardo Coppola, Luca Ardito, Marco Torchiano, and Emil Alégroth. 2020. Translation from Visual to Layout-based Android Test Cases: a Proof of Concept. In *ICST Workshops*. IEEE, 74–83.

[23] Riccardo Coppola, Maurizio Morisio, and Marco Torchiano. 2019. Mobile GUI Testing Fragility: A Study on Open-Source Android Applications. *IEEE Trans. Reliab.* 68, 1 (2019), 67–90.

[24] Luis Cruz, Rui Abreu, and David Lo. 2019. To the attention of mobile software developers: guess what, test your app! *Empirical Software Engineering* 24, 4 (2019), 2438–2468.

[25] Ermira Daka, José Campos, Gordon Fraser, Jonathan Dorn, and Westley Weimer. 2015. Modeling readability to improve unit tests. In *ESEC/SIGSOFT FSE*. ACM, 107–118.

[26] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. 2020. Time-travel testing of Android apps. In *ICSE*. ACM, 481–492.

[27] Marcelo Medeiros Eler, José Miguel Rojas, Yan Ge, and Gordon Fraser. 2018. Automated Accessibility Testing of Mobile Apps. In *ICST*. IEEE Computer Society.

[28] Mattia Fazzini, Eduardo Noronha de A. Freitas, Shauvik Roy Choudhary, and Alessandro Orso. 2017. Barista: A Technique for Recording, Encoding, and Running Platform Independent Android Tests. In *ICST*. IEEE Computer Society, 149–160.

[29] Mattia Fazzini and Alessandro Orso. 2017. Automated cross-platform inconsistency detection for mobile apps. In *ASE*. IEEE Computer Society, 308–318.

[30] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI testing of Android applications via model abstraction and refinement. In *ICSE*. IEEE / ACM, 269–280.

[31] Mark Harman, Lin Hu, Robert M. Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. 2004. Testability Transformation. *IEEE Trans. Software Eng.* 30, 1 (2004), 3–16.

[32] R. Jabbarvand, J. Lin, and S. Malek. 2019. Search-Based Energy Testing of Android. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 1119–1130.

[33] Mona Erfani Joorabchi, Ali Mesbah, and Philippe Kruchten. 2013. Real Challenges in Mobile App Development. In *ESEM*. IEEE Computer Society, 15–24.

[34] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F. Bissyandé, and Jacques Klein. 2019. Automated Testing of Android Apps: A Systematic Literature Review. *IEEE Trans. Reliab.* 68, 1 (2019), 45–66.

[35] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2019. Humanoid: A Deep Learning-Based Approach to Automated Black-box Android App Testing. In *ASE*. IEEE, 1070–1073.

[36] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: an input generation system for Android apps. In *ESEC/SIGSOFT FSE*. ACM, 224–234.

[37] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: multi-objective automated testing for Android applications. In *ISSTA*. ACM.

[38] Stas Negara, Naeem Esfahani, and Raymond P. L. Buse. 2019. Practical Android test recording with espresso test recorder. In *ICSE (SEIP)*. IEEE / ACM, 193–202.

[39] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. 2020. Reinforcement learning based curiosity-driven testing of Android applications. In *ISSTA*. ACM, 153–164.

[40] Anshuman Rohella and Shingo Takada. 2018. Testing Android Applications Using Multi-Objective Evolutionary Algorithms with a Stopping Criteria. In *SEKE*. KSI Research Inc. and Knowledge Systems Institute Graduate School, 308–307.

[41] Kabir S. Said, Liming Nie, Adekunle Akinjobi Ajibode, and Xueyi Zhou. 2020. GUI testing for mobile applications: objectives, approaches and challenges. In *Internetware*. ACM, 51–60.

[42] Leon Sell, Michael Auer, Christoph Frädrich, Michael Gruber, Philemon Werli, and Gordon Fraser. 2019. An Empirical Evaluation of Search Algorithms for App Testing. In *ICTSS (LNCS, Vol. 11812)*. Springer.

[43] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *ESEC/SIGSOFT FSE*. ACM.

[44] Ting Su, Jue Wang, and Zhendong Su. 2021. Benchmarking automated GUI testing for Android against real-world bugs. In *ESEC/SIGSOFT FSE*. ACM, 119–130.

[45] Hongyin Tang, Guoquan Wu, Jun Wei, and Hua Zhong. 2016. Generating test cases to expose concurrency bugs in Android applications. In *ASE*. ACM, 648–653.

[46] Jue Wang, Yanyan Jiang, Chang Xu, Chun Cao, Xiaoxing Ma, and Jian Lu. 2020. ComboDroid: generating high-quality test inputs for Android apps via use case combinations. In *ICSE*. ACM, 469–480.

[47] Wenyu Wang, Dengfeng Li, Wei Yang, Yurui Cao, Zhenwen Zhang, Yuetang Deng, and Tao Xie. 2018. An empirical study of Android test generation tools in industrial cases. In *ASE*. ACM, 738–748.

[48] Haibing Zheng, Dengfeng Li, Beihai Liang, Xia Zeng, Wujie Zheng, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. 2017. Automated Test Input Generation for Android: Towards Getting There in an Industrial Case. In *ICSE-SEIP*. IEEE Computer Society, 253–262.