

# Brewing Up Reliability: Espresso Test Generation for Android Apps

Iván Arcuschin<sup>\*†</sup>, Lisandro Di Meo<sup>\*</sup>, Michael Auer<sup>‡</sup>, Juan P. Galeotti<sup>\*†</sup> and Gordon Fraser<sup>‡</sup>

<sup>\*</sup>University of Buenos Aires, Buenos Aires, Argentina

<sup>†</sup>Institute of Computer Science UBA-CONICET, Buenos Aires, Argentina

<sup>‡</sup>University of Passau, Passau, Germany

{iarcuschin, ldimeo, jgaleotti}@dc.uba.ar {m.auer, gordon.fraser}@uni-passau.de

**Abstract**—The ESPRESSO testing framework for ANDROID has gained popularity among developers as it allows to write concise and reliable UI tests. State-of-the-art tools for automatically testing ANDROID apps, however, tend to produce crash reports rather than human-readable tests, and even if they produce tests these (1) rarely use the ESPRESSO format; (2) are often unreliable due to the volatile nature of widget identifiers; and (3) usually contain no test assertions to serve as regression oracles. While the lack of ESPRESSO support of test generation tools has been addressed by reverse engineering ESPRESSO tests, the other problems remain even with this workaround. In this paper, we therefore introduce a novel ESPRESSO-based representation that allows test generators to generate ESPRESSO test cases directly that (1) can reliably identify widgets using clear and concise ESPRESSO selectors, and (2) can check test executions using ESPRESSO assertions. Experiments on 1,035 ANDROID apps demonstrate that the proposed approach generates ESPRESSO tests that are significantly more reliable than reverse engineered tests, and the ESPRESSO assertions of the generated tests are effective at detecting faults in ANDROID apps.

**Index Terms**—ANDROID test generation, ESPRESSO test cases, AndroZoo benchmark, Mutation analysis

## I. INTRODUCTION

ESPRESSO [1] is a testing framework that helps developers write concise, reliable, and human-readable ANDROID UI tests, and it is the only UI testing framework with substantial adoption amongst app developers [2]. Its popularity is supported by its inclusion in the ANDROID Software Development Kit (SDK), its ability to mitigate flakiness, and the simplicity of the creation and maintenance of tests. Despite this popularity, evidence suggests that ANDROID developers tend to neglect creating automated tests [2], [3].

To aid developers, various automated test generation tools have been introduced [4]–[8]. There are, however, some common limitations to these tools: First, many ANDROID testing tools do not produce *test cases* at all but instead report *crashes*. While this is certainly valuable information, it does not support the aim of building strong regression test suites. Second, even when the tools produce tests, these tests are often (1) unreliable due to the volatile nature of widget identifiers in ANDROID, leading to errors during re-execution; (2) not easy to read or maintain as they rarely use the elegant ESPRESSO framework; and (3) do not contain test assertions, which are a prerequisite for making tests useful during regression testing.

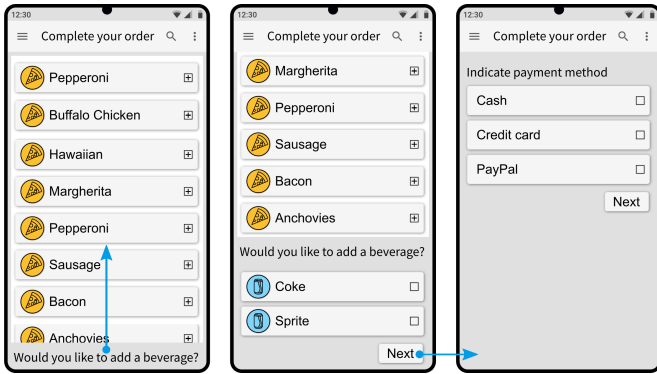
The ESPRESSO framework’s popularity makes it a good candidate as an output format for test generation tools, addressing these concerns. Since most existing test generation tools do not support ESPRESSO, a workaround that has been proposed is to reverse engineer their output, translating it into the ESPRESSO format [9]. However, this *translation-based approach* is inherently challenged by the absence of unique properties that would allow to unambiguously identify UI widgets. This problem is exacerbated as many test generation tools use ANDROID’s Accessibility Service API [10] for collecting screen state information, which may result in inconsistencies [9] such as inaccurate class names of views, missing content description properties, or incorrect text casing. As a result tests may erroneously fail upon re-execution.

In this work, we propose a novel technique to overcome these difficulties: Rather than relying on ANDROID’s Accessibility Service, we introduce a novel representation of test cases for automated test generators, which directly uses the ESPRESSO framework to interact with an app under test (AUT) by defining ESPRESSO selectors that concisely locate ANDROID widgets. This provides several advantages addressing the above listed concerns: First, using the ESPRESSO format matches what developers want. Second, this approach increases the reliability of the generated tests. Third, unlike previous test generation approaches for ANDROID this allows adding test assertions using the ESPRESSO View Assertion mechanism. We implemented this approach in ESPRESSO-MAKER, an extension of the search-based test generation framework MATE [11].

In detail, the contributions of this article are:

- We introduce an approach for generating ESPRESSO tests using ESPRESSO selectors.
- We introduce an approach for generating regression assertions using the ESPRESSO assertions mechanism.
- We provide an open source implementation of the proposed approach in ESPRESSO-MAKER, an extension to the MATE tool for ANDROID test generation.
- We empirically study ESPRESSO-MAKER on 1,035 ANDROID apps with respect to reliability and fault detection.

The experiments demonstrate that the proposed approach increases reliability: 74.33% of the tests generated using ESPRESSO-MAKER can be executed reliably, whereas the traditional translation-based approach leads to only 24.11% re-



(a) 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.

Fig. 1: ANDROID UI example and associated XML code.

executable tests. The study also confirms that the ESPRESSO assertions generated by ESPRESSOMAKER are statistically better at detecting faults than tests without assertions.

ESPRESSOMAKER’s implementation is open-source and can be found online [12], [13]. Additionally, we provide a publicly available replication package [14] containing the source code of ESPRESSOMAKER, the scripts used to run the experiments, and the raw data collected during the empirical study.

## II. BACKGROUND


### A. The ANDROID User Interface

ANDROID app development involves defining separate user interfaces through *Activities*. Each Activity represents a single window for user interaction, composed of a graphical interface and relevant code to respond to both user-triggered events (e.g., button clicks, text field input, etc.) and system-level events (e.g., low battery, network connectivity issues, etc.)

An Activity’s UI can be defined using either code or XML files (as shown in Figure 1b). In both cases, developers must provide a *View Hierarchy* to be interpreted by the ANDROID operating system for rendering the screen. A View Hierarchy consists of a tree-like structure with a root *view* at the top and child views branching off. The *View* class is the basic building block of the ANDROID UI, taking up a rectangular space on the screen and handling both drawing and event handling. It serves as the foundation for interactive UI elements such as buttons and text fields (informally known as *widgets*). Users interact with the mobile app through actions on the visible

```
@Test
public void clickButtonTest() {
    // Build a matcher to find a specific button in the UI.
    Matcher<View> matcher = allOf(
        withId(R.id.button),
        withParent(withId(R.id.layout))
    );
    // Build an interaction to perform actions/assertions
    // on the button.
    ViewInteraction viewInteraction = onView(matcher);
    // Build an action to click the button and execute it.
    ViewAction action = click();
    viewInteraction.perform(action);
    // Build an assertion to check the text of the button
    // and execute it.
    ViewAssertion assertion = matches(withText("Clicked"));
    viewInteraction.check(assertion);
}
```

Fig. 2: Example of an ESPRESSO test. This test builds an ESPRESSO View Matcher for selecting a view that has, at the same time, a resource identifier “R.id.button” and a parent with resource identifier “R.id.layout”. It then clicks on the selected view and asserts that its text is “Clicked”.

views. For example, clicking the “Add item” ImageButton (i.e., the  button) in the rows of Figure 1a. Lastly, internal nodes in the View Hierarchy are *View Groups*, which are views that can contain other views. An example of a UI and part of its hierarchy is shown in Figure 1.

### B. Testing ANDROID Apps

JUNIT [15] is a widely used framework for JAVA unit testing. In ANDROID projects, pure JUNIT test cases can only be used to exercise classes that have no interactions with the ANDROID framework. Testing the GUI components of an app, such as activities and views, requires *instrumented tests*, which run on ANDROID devices, either physical or emulated [16], allowing them to take advantage of the ANDROID framework APIs. These tests are initialized in a special environment that gives them access to an instance of the Instrumentation class [17], which allows developers to monitor all the interactions the ANDROID system has with the AUT. Instrumented tests (also known as *instrumentation tests*) provide more fidelity than unit tests, although they run much more slowly.

### C. The ESPRESSO Testing Framework

Several testing frameworks have been proposed to help ANDROID developers in writing automated tests, such as APIUM [18], CALABASH [19], ESPRESSO [1], MONKEYRUNNER [20], ROBOTIUM [21], and UIAUTOMATOR [22]. Of these, ROBOTIUM and CALABASH are no longer maintained. ESPRESSO [1] is an *Instrumentation*-based UI testing framework, officially supported by the ANDROID ecosystem and part of the AndroidX Test Repository [23]. This, and the framework’s ability to provide concise and reliable UI tests, has made it the most popular testing framework among developers [2], with a recent steady increase in adoption.

ESPRESSO provides an API that allows developers to programmatically simulate user interactions with the AUT. The framework also enables developers to specify which ANDROID activity should be launched at the start of each test using a @Rule annotation. Furthermore, ESPRESSO enhances test reliability by executing pending actions only when the AUT is idle. Creating a reliable ESPRESSO test case involves writing:

- *View Matchers* to identify and select target views within the current View Hierarchy. These matchers need to be specific. If the view is not found or multiple views match the criteria in a given screen, the test execution will fail. For example, a matcher that locates a TextView by its text will fail if the text is not unique in the UI.
- *View Actions* to perform actions on selected views. These actions need to be adequate for the targeted views. For example, a *click* action will fail if it is executed on a view that is not displayed on the screen.
- *View Assertions* to verify the state (properties) of selected views. The properties to check need to be valid for the targeted view. For example, ImageView UI elements do not have text, so using the *withText* assertion on such views leads to an exception.

Figure 2 displays an ESPRESSO test as example.

#### D. Generating ANDROID Test Cases with MATE

Most test generation tools for ANDROID use the ANDROID Accessibility Service [10] to retrieve information and interact with an app. For years, ANDROID’s Accessibility Service was the only official API that provided a way to retrieve information about the UI and to interact with it. However, the main purpose of the Accessibility Service’s API is to assist users with disabilities in using ANDROID devices and apps, and it is not designed for testing purposes. In particular, it can return inconsistent information [9] such as imprecise *class names* of views, incorrectly reporting the *hint* of a field as *text* input, missing *content description* properties, and providing texts with the wrong casing.

Although in principle the question of how to generate ESPRESSO test cases is orthogonal to the question of what test generation approach or tool to use, a proof of concept requires adapting a concrete test generator to use ESPRESSO APIs rather than the ANDROID Accessibility Service. We chose the MATE [11] test generation tool in order to generate sequences of interactions for ANDROID apps, and adapted it to use ESPRESSO. Though MATE was originally designed to find accessibility problems (e.g., missing content descriptions for visible components), it has recently been extended to study different evolutionary algorithms for test generation [24].

Internally, each test case in MATE is represented as a sequence of actions on available views of the AUT. MATE outputs an XML representation of the test cases, and a report on the number of executed test cases, found crashes, and achieved coverage. In order to generate test cases, MATE (as well as other ANDROID test generators such as STOAT or DYNODROID) requires to perform the following tasks:

TABLE I: View Actions used in ESPRESSOMAKER.

ESPRESSO action	Description
pressBack()	Clicks the “back” button.
clearText()	Clears text on the view.
click()	Clicks the view (i.e., single tap).
pressKey(ENTER)	Presses the hardware <i>enter</i> key.
longClick()	Long clicks the view (i.e., long single tap).
pressMenuKey()	Presses the hardware <i>menu</i> key.
pressKey(SEARCH)	Presses the hardware <i>search</i> key.
swipeDown()	Performs a swipe top-to-bottom across the horizontal center of the view.
swipeLeft()	Performs a swipe right-to-left across the vertical center of the view.
swipeRight()	Performs a swipe left-to-right across the vertical center of the view.
swipeUp()	Performs a swipe bottom-to-top across the horizontal center of the view.
typeText(String)	Selects the view (by clicking on it) and types the provided string into the view.

TABLE II: View Matchers used in ESPRESSOMAKER

ESPRESSO matcher	Description
isRoot()	The view is the root of the View Hierarchy.
withId(Integer)	The view has a specific id.
withResourceName(String)	The view has a specific resource name.
withText(String)	The view has a specific text.
withContentDesc.(String)	The view has a specific content description.
withClassName(String)	The view has a specific class name.
allOf(ViewMatcher...)	The view matches against <i>all</i> specified matchers.
withChild(ViewMatcher...)	The view matches only if one the view’s direct child views matches against specified matchers.
withParent(ViewMatcher...)	The view matches only if its parent view matches against specified matchers.

- Given the current state of the app, retrieve the list of all available actions.
- Given a chosen available action on the app, perform the selected action.
- (Optionally) Having executed an action, contrast the expected state of the app against the actual observed state.

All these tasks are conducted independently of the underlying algorithm for generating the test cases. In other words, it is possible to identify these tasks in search-based test generators (e.g., MATE), model-based test generators (e.g., STOAT), or probabilistic generators (e.g., DYNODROID).

### III. GENERATING ESPRESSO TESTS

A fundamental task in automated test generation is collecting available actions for an app in its current state. Given the current screen, we refer as *screen information* to:

- The View Hierarchy of the current screen, which also includes the properties of each view such as *resource identifier*, *text*, and *content description*.
- The available ESPRESSO actions for each view in the View Hierarchy.

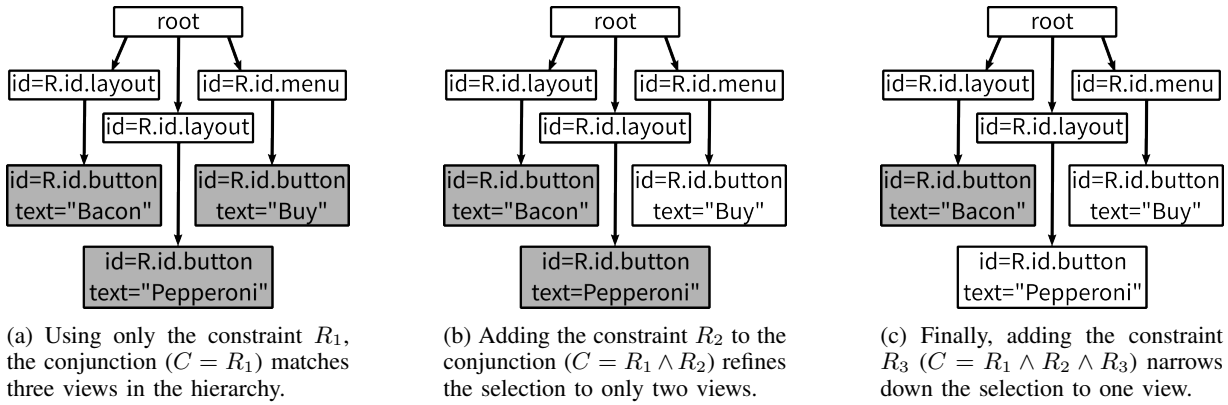


Fig. 3: Iteratively combining View Property Constraints to locate a target view in the View Hierarchy. Each box represents a view in the View Hierarchy. Gray boxes indicate the views that match against the conjunction of constraints, while white boxes indicate the views that do not match. The constraints used in this example are:  $R_1 = WITH\_ID(v, R.id.button)$ ,  $R_2 = WITH\_ID(v.parent, R.id.layout)$  and  $R_3 = WITH\_TEXT(v, \text{"Bacon"})$

- The package and activity name of the app being shown on the screen. This is needed to know when a test generator has “exited” the AUT.

Unfortunately, the ESPRESSO framework does not provide direct access for retrieving the first two items. Nevertheless, we can still access the View Hierarchy through reflection [25].

To obtain the actions that are available for a given view, we iterate over all ESPRESSO actions listed in Table I. We filter this list by checking for each action if it can be performed on the view by calling the `getConstraints` method on the action. This method allows View Actions to specify constraints that the views have to satisfy in order to be considered as targets for that action. For example, the `click()` action requires the view to be shown on the screen (i.e., `isDisplayed()`).

#### A. Generating ESPRESSO View Matchers

Once the available views and actions have been determined, the next step consists of creating an ESPRESSO View Matcher for each targeted view using this information. View Matchers are crucial since they serve the dual purpose of allowing execution of View Actions and View Assertions. These View Matchers must unequivocally locate the targeted view. A View Matcher is *unequivocal* if there is one, and only one, view in the View Hierarchy that matches against it. If the View Matcher targets none or more than one view, the test execution will end in a failure.

Constructing View Matchers for a targeted view might not always be feasible due to the lack of unique properties to unambiguously locate the view. This problem is mentioned in the related literature as *view disambiguation* [9], [26]–[28]. View Matchers are defined using resource identifiers along other view properties (such as text, children and parent views, etc.) Often, the targeted view has a unique resource identifier which can be used to unequivocally locate it (i.e., using the ESPRESSO `withId` matcher). However, there are many legitimate scenarios in which a targeted view may not have a resource identifier or this identifier might not be unique. As

an example, consider the scenario in which the rows of a list are rendered using an XML document (e.g., Fig. 1b). In such a scenario, all rows share the same resource identifier.

Given a target view, our technique for generating an unequivocal ESPRESSO View Matcher starts by identifying a set of constraints that are only satisfied by the target view; i.e., these constraints must not be satisfied by any other views in the View Hierarchy. For example, in Fig. 2 the target view  $v$  satisfies the constraint  $v.resourceIdentifier == R.id.button$  while at the same time satisfying the constraint  $v.parent.resourceIdentifier == R.id.layout$ .

In each of these constraints we can find the following elements: there is a view that can be accessed by navigating the View Hierarchy starting from the target view (e.g.,  $v$ , and  $v.parent$ ), and there is a view property (e.g., resource identifier) of the navigated view that has to match a given literal value (e.g.,  $R.id.button$ , and  $R.id.layout$ ).

We define a *View Property Constraint* as a constraint on a property of a view  $v$  using a value  $c$ . For example, the constraint  $WITH\_ID(v, c)$  is defined as  $v.resourceIdentifier == c$ . Notice that  $v.resourceIdentifier == R.id.button$  can be rewritten as  $WITH\_ID(v, R.id.button)$ . Then, the View Matcher in Fig. 2 for the target view  $v$  can be expressed as a conjunction  $R_1(v) \wedge R_2(v)$  of view property constraints, such that:

$$R_1(v) = WITH\_ID(v, R.id.button)$$

$$R_2(v) = WITH\_ID(v.parent, R.id.layout)$$

$R_1(v)$  states that the targeted view must have the value of  $R.id.button$  as resource identifier. Additionally,  $R_2(v)$  states that the targeted view’s parent must have the value of  $R.id.layout$  as resource identifier. Although  $R_1(v)$  and  $R_2(v)$  are both parameterized by the target view  $v$ , the view on which the constraint is checked differs in each case. In  $R_1(v)$  the constraint is checked on the target view  $v$ , while in  $R_2(v)$  the constraint is checked on  $v$ ’s parent view. Thus, when the target view is fixed (e.g., a given  $v$ ) we directly use the “traversal” path as the first argument of the view property constraint.

TABLE III: View Assertions used in ESPRESSOMAKER.

ESPRESSO assert.	Description
doesNotExist	The view does not exist in the View Hierarchy.
isDisplayed	The view is displayed (and visible) at least 90% on the screen.
isEnabled	The view is enabled. The interpretation of the enabled state varies by subclass.
isNotEnabled	The view is not enabled.
isFocused	The view is focused (i.e., the user is directly interacting with the view).
isNotFocused	The view is not focused.
isFocusable	The view is focusable (i.e., it can receive the focus).
isNotFocusable	The view is not focusable.
hasFocus	The view has focus. This means that the view or one of its descendants is focused.
doesNotHaveFocus	The view does not have focus.
isSelected	The view is selected. This is not the same as focus, it refers to the selected status in the context of a list or similar (e.g., view is highlighted).
isNotSelected	The view is not selected.
isChecked	The view is checked. This can only be true for “checkable” views such as a CheckBox.
isNotChecked	The view is not checked.
isClickable	The view is clickable (i.e., the view accepts and reacts to click/tap events).
isNotClickable	The view is not clickable.
hasLinks	The view is a <code>TextView</code> and contains URLs.
hasContentDesc.	The view’s content description is not null (can be empty String).
withText	The view has a specific text.
hasErrorText	The view has a specific error text.
withContentDesc.	The view has a specific content description.
withHint	The view has a specific hint.
withAlpha	The view has a specific opacity. This is a value from 0 to 1, where 0 means the view is completely transparent and 1 means the view is completely opaque.
hasChildCount	The view has a specific number of children in the View Hierarchy. This is value is always 0 if the view is not a <code>ViewGroup</code> .
withInputType	The view is an <code>Editable</code> object and has a specific type of basic content (e.g., text, number, passwords, etc.).
withParentIndex	The view is a child with the specific index in its parent.
withVisibility	The view has a specific <code>Visibility</code> state. Possible values are <code>Visible</code> , <code>Invisible</code> , and <code>Gone</code> . <code>Invisible</code> means that the view is not visible, but it still takes up space for layout purposes. <code>Gone</code> means that the view is invisible and it does not take up any space.

In summary, the view property constraints represent the building blocks that will be later used for building the actual ESPRESSO View Matcher. If the conjunction of constraints is *unequivocal* there will be one and only one view in the hierarchy that matches against all constraints simultaneously. Thus, by iteratively adding constraints to a conjunction we aim to build an expression that unequivocally identifies a target view in the View Hierarchy. This is illustrated in Fig. 3.

Algorithm 1 outlines the steps for generating unequivocal *View Property Constraint* conjunctions. This algorithm starts by checking if the input view  $v$  is the root of the View Hierarchy  $t$ . In that case, we simply return the `IS_ROOT` constraint. Otherwise, we iterate over all views in the View Hierarchy  $t$  (including  $v$  itself). For each view  $v'$  in  $t$ , we

**Algorithm 1:** Algorithm for generating an unequivocal *View Property Constraint* conjunction

---

```

Input : View  $v$ , View hierarchy tree  $t$ 
Output : View Property Constraints conjunction  $C$ 
1 if  $v$  does not have parent do
2    $\lfloor$  return IS_ROOT( $v$ )
3  $C \leftarrow \text{True}$ 
4 for View  $v' \in t$  do
5    $path \leftarrow \text{PATHTOVIEW}(v, v', t)$ 
6   for Constraint  $P \in \text{BASIC\_CONSTRAINT\_TYPES}$  do
7     if type is valid for  $v'$  do
8        $value \leftarrow \text{GETVALUE}(v', P)$ 
9        $C \leftarrow C \wedge P(path, value)$ 
10      if  $C$  is unequivocal do
11         $\lfloor$  return GETMINIMALCOMBINATION( $C$ )
12 return Null

```

---

determine the path that needs to be traversed in  $t$  from  $v$  to  $v'$ . For example, a navigation path might state that in order to get from  $v$  to  $v'$ , we need to go “up” to  $v$ ’s parent view and then go “down” to the parent’s first child. We add one constraint to  $C$  for each *basic constraint type* (namely, resource name, resource identifier, text, content description and class name). Each constraint is only added if the property that it checks is defined for  $v'$ . For example, a text constraint will not be added for a view without text. The value that the constraints will match against is the actual value observed for each property in  $v'$ . Finally,  $C$  is returned once we find an *unequivocal* conjunction. If no such conjunction exists, we return `Null` and omit  $v$  from the available actions for the current screen.

Once an *unequivocal View Property Constraint* conjunction is found, we *minimize* it. A conjunction is *minimal* when removing any constraint leads to matching against more than one view in the View Hierarchy. This minimization is performed by means of Delta Debugging [29].

Once we have successfully found a *minimized unequivocal View Property Constraint* conjunction, we translate it into a corresponding ESPRESSO View Matcher. Algorithm 2 describes the steps taken to perform this task. The algorithm starts by creating an “*AllOf*” View Matcher: a recursive matcher that selects a view only if it matches against all the specified inner matchers. For each *View Property Constraint* in the conjunction, the `AppendInnerMatcher` function recursively builds the ESPRESSO View Matcher that satisfies the constraint’s path and type, and appends it where necessary.

As an example, consider the *View Property Constraint* conjunction  $R_1(v) \wedge R_2(v)$  such that

$$R_1(v) = \text{WITH\_ID}(v.parent, R.id.button)$$

$$R_2(v) = \text{WITH\_TEXT}(v.parent, "Bacon")$$

First, the algorithm executes `AppendInnerMatcher` on  $R_1$ , producing the partial ESPRESSO View Matcher

$$\text{allOf}(\text{withParent}(\text{withId}(R.id.button)))$$

---

**Algorithm 2: Generation of ESPRESSO View Matcher**

---

```
Input : View  $v$ , View Property Constraint conjunction  $C$ 
Output : ESPRESSO View Matcher  $M$  for selecting  $v$ 
1 Function BUILDESPRESSOVIEWMATCHER( $v, C$ )
2    $M \leftarrow \text{ALLOF}(\{\})$ 
3   for Constraint  $c \in C$  do
4      $type \leftarrow \text{GETTYPE}(c)$ 
5      $path \leftarrow \text{GETPATH}(c)$ 
6     APPENDINNERMATCHER( $M, v, type, path$ )
7   return  $M$ 
8 Function APPENDINNERMATCHER( $M, v, type, path$ )
9   if  $path$  is empty do
10    if  $type = \text{WITH\_RESOURCE\_NAME}$  do
11       $newMatcher \leftarrow$ 
12        WITHRESOURCENAME(GETRESOURCENAME( $v$ ))
13    if  $type = \text{WITH\_ID}$  do
14       $newMatcher \leftarrow \text{WITHID}(\text{GETID}(v))$ 
15    if  $type = \text{WITH\_TEXT}$  do
16       $newMatcher \leftarrow \text{WITHTEXT}(\text{GETTEXT}(v))$ 
17    if  $type = \text{WITH\_CONTENT\_DESCRIPTION}$  do
18       $newMatcher \leftarrow$ 
19        WITHCONTENTDESCRIPTION(GETCONTDESC( $v$ ))
20    if  $type = \text{WITH\_CLASS\_NAME}$  do
21       $newMatcher \leftarrow$ 
22        WITHCLASSNAME(GETCLASSNAME( $v$ ))
23     $M.matchers.APPEND(newMatcher)$ 
24  else
25     $nextStep \leftarrow \text{HEAD}(path)$ 
26    if  $nextStep = \text{MOVE\_TO\_PARENT}$  do
27      if  $M.matchers$  does not contain a
28        WithParent matcher do
29         $newMatcher \leftarrow \text{WITHPARENT}()$ 
30         $M.matchers.APPEND(newMatcher)$ 
31      else
32         $newMatcher \leftarrow$ 
33        GETPARENTMATCHER( $M.matchers$ )
34    else
35      if  $M.matchers$  does not contain a WithChild
36        matcher do
37         $newMatcher \leftarrow \text{WITHCHILD}()$ 
38         $M.matchers.APPEND(newMatcher)$ 
39      else
40         $newMatcher \leftarrow$ 
41        GETCHILDMATCHER( $M.matchers$ )
42     $remainingPath \leftarrow \text{TAIL}(path)$ 
43     $w \leftarrow \text{GETVIEWAFTERSTEP}(v, nextStep)$ 
44    ADDMATCHERFORPATHANDTYPE( $newMatcher, w,$ 
45       $type, remainingPath$ )
46  return  $M$ 
```

---

Then, the algorithm executes AppendInnerMatcher on  $R_2$ , producing the final ESPRESSO View Matcher:

```
allOf( withParent( allOf(
  withId(R.id.button), withText("Bacon"))) ) )
```

Table II lists all the ESPRESSO View Matchers used by the algorithm.

---

**Algorithm 3: ESPRESSO View Assertion Generation**

---

```
Input : Test case  $tc$ , Assertion level  $L$ 
Output : List of assertions  $A$ 
1  $A \leftarrow \{\}$ 
2 if  $L == \text{None}$  do
3   return  $A$ 
4 Start AUT
5  $lastScreen \leftarrow \emptyset$ 
6 for Action  $a \in tc$  do
7   EXECUTEACTION( $a$ )
8    $currentScreen \leftarrow \text{GETSCREENINFO}()$ 
9   if  $L == \text{Full} \vee (L == \text{SemiFull} \wedge lastScreen == \emptyset)$ 
10    do
11      for View  $v \in \text{GETALLVIEWS}(currentScreen)$  do
12        for  $attr \in \text{GETATTRIBUTES}(v)$  do
13           $A.APPEND($ 
14            BUILDATTRIBUTEASSERTION( $v, attr$ )
15          )
16      else
17         $screenDiff \leftarrow$ 
18        COMPUTESCREENDIFF( $currentScreen, lastScreen$ )
19        for View  $v \in \text{GETCOMMONVIEWS}(screenDiff)$  do
20          for  $attr \in \text{GETATTRIBUTESDIFF}(screenDiff, v)$ 
21          do
22             $A.APPEND($ 
23              BUILDATTRIBUTEASSERTION( $v, attr$ )
24            )
25        for View
26           $v \in \text{GETDISAPPEARINGVIEWS}(screenDiff)$  do
27           $A.APPEND(\text{DOESNOTEXIST}(v))$ 
28        for View  $v \in \text{GETAPPEARINGVIEWS}(screenDiff)$  do
29           $A.APPEND(\text{ISDISPLAYED}(v))$ 
30          if  $L == \text{SemiFull}$  do
31            for  $attr \in \text{GETATTRIBUTES}(v)$  do
32               $A.APPEND($ 
33                BUILDATTRIBUTEASSERTION( $v, attr$ )
34              )
35         $lastScreen \leftarrow currentScreen$ 
36  return  $A$ 
```

---

### B. Generating ESPRESSO View Assertions

In software testing, an assertion is a statement that checks if a specific property of the program state matches its expected value. If the actual value differs, the assertion fails the test case. The ESPRESSO framework employs assertions to ensure that a selected view of the UI is in the desired state.

We present an assertion generation algorithm for ESPRESSO (Algorithm 3) based on Xie's algorithm [30], which consists of tracing the values of all observable attributes after each step of a test case, and then inserting regression assertions that capture these values. Intuitively, the conventional approach of generating assertions for *all* attributes of all views and then filtering non-relevant assertions through mutation analysis [31] is unlikely to be feasible for ESPRESSO assertion generation due to the high number of views and attributes in an ANDROID app, and the costly execution time of ANDROID test cases. Algorithm 3 therefore executes each test action one at a time, determining at each step which assertions to add to the test case based on the current and previous View Hierarchies, using

the observed values as expected values in the assertions. It can be configured in four different levels of assertion generation: *Full*, *SemiFull*, *DiffOnly*, and *None*:

- The *Full* level generates assertions for all attributes of all views in the View Hierarchy (i.e., ignoring the previous View Hierarchy). While this likely results in large numbers of (possibly non-relevant) assertions, it serves as a baseline for comparison with the other levels.
- The *DiffOnly* level generates assertions only for attributes (such as text) that have changed in common views between consecutive screens. It also adds assertions when views appear (or disappear) from the UI. The rationale behind this level is that the most relevant assertions are those that check for parts of the UI that are dynamically updated. This approach significantly reduces the number of assertions generated, but it can also result in missing assertions for attributes that do not change across the execution of a test case.
- The *SemiFull* level aims to mitigate this issue by not only asserting attributes of common views between consecutive screens, but also of *appearing* views (i.e., views present in a screen but absent in the screen immediately before). This level then adds assertions for attributes that have changed in common views (same as *DiffOnly*), and assertions for all attributes of appearing views.
- Finally, the *None* level does not generate any assertions.

Table III lists the ESPRESSO View Assertions generated by the algorithm.

#### IV. EVALUATION

In order to investigate the necessity and effectiveness of creating ESPRESSO tests, we aim to empirically answer the following research questions:

**RQ1(Motivation):** *Are resource identifiers sufficiently unique for building ESPRESSO View Matchers?*

**RQ2(Reliability):** *Are the ESPRESSO tests generated by ESPRESSOMAKER more reliable than the ones generated with a translation-based approach?*

**RQ3(Assertions):** *Are the ESPRESSO assertions generated by ESPRESSOMAKER effective at detecting faults?*

##### A. ESPRESSOMAKER’s Implementation

The algorithms presented in Section III were implemented as an extension of MATE. We refer to this extension as ESPRESSOMAKER throughout the article. MATE’s original implementation uses an *Instrumentation*-based process to communicate with ANDROID’s Accessibility Service API for collecting the AUT UI’s current state. In this original setup, the AUT itself is run as a separate process without any instrumentation. However, in order to use the ESPRESSO framework within MATE, the latter needs to run in the same process as the AUT. We refactored MATE’s architecture to include a new *Representation Layer* that runs in the instrumented AUT process, employs the ESPRESSO framework internally, and informs the View Hierarchy and available actions to MATE’s exploration module. Since both modules now run in

separate processes, they rely on the Android Interface Definition Language (AIDL) [32] for interprocess communication using Remote Procedure Calls (RPC). ESPRESSOMAKER’s implementation is publicly available on GitHub [12], [13].

##### B. Experimental Setup

*Selection of Subjects Under Test:* We evaluate ESPRESSOMAKER on a subset of the AndroZoo dataset [33]. We initially downloaded 7,000 APKs randomly chosen from this dataset and then performed a health check on each app to exclude those that were non-functional or incompatible with our experimental setup. We discarded 2,946 apps that were not compatible with the x86 emulator architecture, and 1,228 apps that crashed when running the MONKEY tool [34] on them for 1 minute. Additionally, we removed 1,791 apps that could not be instrumented by MATE’s bytecode instrumentation, also used in ESPRESSOMAKER, which is needed to measure code coverage on APKs (AndroZoo does not provide apps’ source code). This process resulted in 1,035 APKs.

*Experiment Procedure:* We used ESPRESSOMAKER’s implementation to generate ESPRESSO tests for the 1,035 apps in our dataset. We chose MATE’s *Random Exploration* algorithm for generating test cases, which has been shown to be one of the best performing algorithms for ANDROID test generation [24]. We limit the generation budget to 10 test cases (i.e., individuals) per subject, with a maximum of 15 actions (only GUI actions such as click, long click, type text, etc.) allowed for each test case. *Random Exploration* works by continuously sampling the search space until it runs out of budget or the maximum number of actions is reached. The generation of an individual is also halted when the generation algorithm exits the AUT’s UI. On each iteration, a completely new individual is created. The experiments were fully automated, with no manual intervention (e.g., logins) during MATE’s execution. Results for **RQ1** are based on the UI states and actions explored by ESPRESSOMAKER during the test generation process.

For **RQ2**, we implemented the translation-based approach proposed by Arcuschin et al. [9] in MATE. We compare the reliability of the ESPRESSO tests generated by ESPRESSOMAKER against the tests generated by translation. We consider a test to be “*reliable*” if it can be successfully executed (i.e., it does not fail). Note that ESPRESSO tests generated by the translation-based approach do not contain assertions.

For **RQ3**, we consider the AndroZoo APKs from **RQ2** for which we were able to generate at least one passing (non-flaky) ESPRESSO test (926 apps). Since flaky tests may affect mutation analysis results, we re-executed the tests 3 times to identify and remove flaky tests. We employed MUTAPK [35] to generate mutants of these apps. MUTAPK takes as input an APK and produces a set of modified APKs by applying a set of predefined mutation operators. In our experiment, we limited the mutation operators to only those that have the potential to change the app’s UI, since the assertions in ESPRESSO tests are based on the UI state. We discarded the mutation operators leading to trivial mutants (such as the

ActivityNotDefined operator, which can only produce a crash in the AUT) and non-observable mutants (e.g., operators that do not change the AUT’s UI, such as InvalidActivityName). For this process, two of the authors manually inspected each mutation operator available in MUTAPK. When there was disagreement, we reached consensus by discussing the mutation operator. The final operators considered in our experiment were: DifferentActivityIntentDefinition, WrongMainActivity, MissingPermissionManifest, WrongStringResource, InvalidURI, NullGPSLocation, InvalidDate, ViewComponentNotVisible, and InvalidViewFocus. Given the large number of mutations (222 on average per app), we randomly sampled 10 mutants per app. We also considered MUDROID [36] and DROIDMUTATOR [37] for generating mutants. MUDROID’s implementation is no longer maintained and we were unable to generate mutants with it. DROIDMUTATOR requires source code, but the AndroZoo dataset only contains binaries.

The experiments were conducted on a cluster comprising 12 nodes, each equipped with 2 Intel Xeon E5-2650 v2 CPUs (8 cores each) and 128 GB of RAM. We selected ANDROID Lollipop (API 21) as the target platform for our experiments. For measuring statistical significance, we relied on the well-established Wilcoxon-Mann-Whitney U-test and Vargha-Delaney  $A_{12}$  effect size [38].

### C. Threats to Validity

Threats to internal validity might result from how the empirical study was carried out. Our implementation effort was large, so we compared different tools and picked the one that better suited our needs. We chose to extend MATE for implementing ESPRESSOMAKER due to its maturity (actively maintained) and our prior experience working with it. However, the techniques presented in this work are not tied to MATE and can be applied to other test generation tools as long as they execute the generated test cases (e.g., on an emulator or device). We used random exploration since a prior study reported this to achieve best results [24]; the representation should be independent of the algorithm used. To avoid possible confounding factors when comparing ESPRESSOMAKER to the translation-based approach, they were both implemented on the same tool (MATE). Since parameter tuning can affect the performance of algorithms [39], we used the same default values for all parameters across experiments. For RQ3, we aimed to mitigate bias by using the MUTAPK tool for mutation analysis, and randomly selecting 10 mutants for each app. We relied on mutation analysis as a proxy for fault detection. Although mutation analysis is a well-established technique, the artificial mutants might not be representative of actual defects.

Threats to external validity might result from how the selection of subjects was carried out. To mitigate this threat, we randomly downloaded 7,000 APKs from the AndroZoo dataset [33], and only excluded those that were non-functional or incompatible with our experimental setup. Another selection of subjects might yield different results.

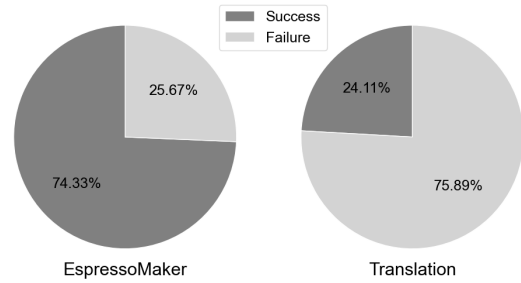


Fig. 4: Percentage of test cases successfully executed for RQ2.

### D. Results RQ1: Are resource identifiers sufficiently unique for building ESPRESSO View Matchers?

ESPRESSOMAKER encountered 4,387,217 views while generating ESPRESSO tests for the 1,035 apps in our dataset. Developers only assigned 61.90% of the views a resource identifier, for the remaining 38.10% the ANDROID platform assigns a default resource identifier of  $-1$ . However, even when developers assign resource identifiers these are not always unequivocal: Only 46.00% of all views had an unequivocal resource identifier assigned, while 15.90% had a non-unequivocal one. Overall,  $38.10\% + 15.90\% = 54.00\%$  of all views either lacked a developer-assigned resource identifier or had a non-unequivocal assigned identifier, thus requiring a non-trivial matcher for performing ESPRESSO actions on them.

ESPRESSOMAKER was able to generate an unequivocal matcher combination for 92.45% of all views encountered. For the remaining 7.55%, we manually inspected the affected apps and found that these failures were caused by identical twin views: sibling views (i.e., children of the same layout parent) with identical attributes, which cannot be disambiguated by adding additional information about their parents (since they have the same parent) or children (because they do not have children views or because the children are also identical). In principle, this issue could be solved by making use of the withParentIndex ESPRESSO View Matcher in Algorithm 2, allowing ESPRESSOMAKER to specify the index of a child in its parent view. We leave this as future work.

**RQ 1:** More than half of the views in our dataset do not have an unequivocal resource ID, requiring a non-trivial matcher to perform an ESPRESSO action on them.

### E. Results RQ2: Are the ESPRESSO tests generated by ESPRESSOMAKER more reliable than the ones generated with a translation-based approach?

ESPRESSOMAKER generated 11,049 ESPRESSO test cases, while the translation-based approach generated 11,255 ESPRESSO test cases; the minor difference was caused by timeouts during ESPRESSOMAKER’s execution. Considering the small number of such cases we believe this demonstrates sufficient robustness of our results. Figure 4 illustrates the test execution results: Among the 11,049 test cases generated by ESPRESSOMAKER, 8,213 (74.33%) were successfully executed, while only 2,714 (24.11%) passed for the translation-



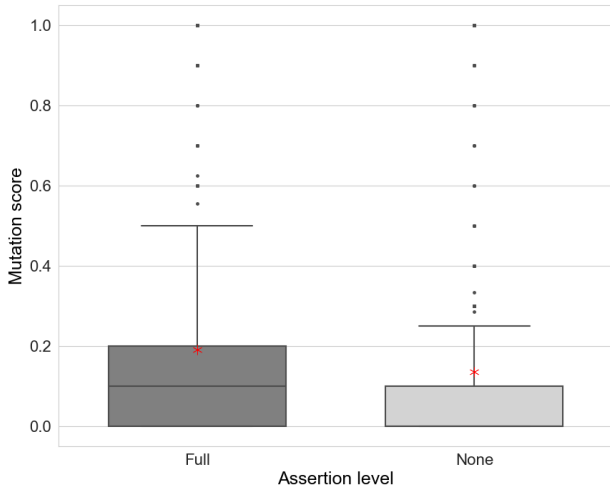


Fig. 5: Mutation scores for *Full* and *None* assertion levels.

based approach. The difference is statistically significant according to a Wilcoxon rank-sum test at  $\alpha = 0.05$  with  $p < 0.001$ , and a large Vargha-Delaney effect size of  $\hat{A}_{12} = 0.82$ .

**RQ 2:** *Tests generated by ESPRESSOMAKER are significantly more reliable than translated tests.*

*F. Results RQ3: Are the ESPRESSO assertions generated by ESPRESSOMAKER effective at detecting faults?*

We compare the mutation scores [40] (i.e., mutants detected over evaluated mutants) of the (8,252 non-flaky) ESPRESSO test cases with *Full* and no assertions (*None*).

Figure 5 illustrates the mutation scores achieved by the ESPRESSO test cases using *Full* and *None* assertion levels, on a total of 8,785 mutants. Test cases with *Full* assertions achieved a mean mutation score of 19.09% (1,703 mutants killed), while those with no assertions achieved 13.45% (1,202 mutants killed). This difference is statistically significant ( $p < 0.001$ ,  $\hat{A}_{12} = 0.57$ ). Notice that the mutants detected by tests cases without assertions are either due to crashes or changes in the UI leading to test errors: In contrast to unit-level testing, UI testing relies on selectors, such as ESPRESSO View Matchers, to locate and interact with UI elements. These selectors are sensitive to UI changes, triggering test errors when the specified conditions are no longer met.

The *Full* assertion level can potentially generate an infeasibly large number of assertions. While the tests without assertions have a fixed length range of 1 to 15 statements, depending on the number of actions performed by the test case this increases to an average of 1,157 statements per test for *Full*. The *DiffOnly* and *SemiFull* approaches aim to reduce this, which they achieve successfully, with 262 statements per test for *SemiFull*, and 29 for *DiffOnly*.

To see whether this reduction of the number of assertions comes at the price of a drop in fault detection effectiveness, we consider the mutant detection ratios: We find that *SemiFull* detected 89.08% of the mutants killed by *Full*, while *DiffOnly* and *None* detected 71.29% and 63.30%, respectively.

Therefore, the different assertion levels introduce a trade-off between the length of test suites and their ability to detect faults. Considering that an unreasonably large number of assertions like that of *Full* would make tests hard to maintain, and considering also that the good fault detection capability of *SemiFull* and *DiffOnly* assertions, using the *SemiFull* level appears to be a good practical compromise.

**RQ 3:** *Tests generated by ESPRESSOMAKER detect significantly more mutants when including assertions.*

To shed more light on these findings, we inspect the results of the mutation analysis in more detail. In particular, the RQ3 results appear to show rather low mutation scores. Note that, in order for a test case to detect a mutant, the following must happen: (1) the mutated part of the program needs to be executed, (2) the executed faulty part needs to change the program state, and (3) the new behavior exhibited by the mutant needs to be observed either with an assertion or a crash.

Unfortunately, it is common for the mutation operators in MUTAPK to work at the XML level (e.g., the *WrongStringResource* operator), which cannot be directly mapped to the source code. This makes it difficult to measure exactly how many mutants were executed (i.e., condition (1) from above) as it is not always clear which part of the app is affected by the mutation, and whether the mutated part is actually executed by the test cases. In our experiments, 85.34% of the mutants were produced by XML mutation operators. We analyzed the remaining 14.66% mutants produced by code mutation operators. By comparing the execution traces of the generated ESPRESSO tests and the location of the modified bytecode in mutants, we found that the mutated code of 65.07% mutants was not reached by any of the test cases during our experiments. Note that our experiment only consisted of sampling a comparatively small set of test cases, rather than applying test generation to fully optimize a test suite for coverage, hence the low coverage of mutants is expected. However, having demonstrated the effectiveness of the assertions in principle, it is clear that increasing the tests' coverage will lead to higher overall mutation scores. Additionally, although we discarded mutation operators leading to trivial or non-observable mutants, some generated mutants may still be equivalent to the original apps; this holds in general for mutation analysis.

To understand the impact of condition (3), we compared the mutants created by MUTAPK against the assertions generated by ESPRESSOMAKER. The top five types of mutants (comprising 98.11% of the total) were: *WrongStringResource* (63.34%), *MissingPermissionManifest* (20.27%), *ViewComponentNotVisible* (5.10%), *InvalidViewFocus* (4.99%), and *DifferentActivityIntentDefinition* (4.41%). Table III shows the different types of assertions generated by the *Full* assertion level. We find that the most common types of mutants are well covered by these assertions: The *withText* assertions tend to detect *WrongStringResource* mutants, the *withVisibility* assertions detect *ViewComponentNotVisible* mutants, and the *hasFocus*, *isFocused* and *isSelected* assertions detect *InvalidViewFocus* mutants. The *isDisplayed* assertions detect

MissingPermissionManifest and DifferentActivityIntentDefinition mutants, which potentially produce larger UI changes, as well as ViewComponentNotVisible mutants. Thus, the assertions generated by ESPRESSOMAKER are well-suited for detecting the UI changes introduced by MUTAPK mutants. Since MUTAPK’s mutation operators are derived from a taxonomy of real ANDROID faults [41], ESPRESSOMAKER’s assertions are likely also useful for detecting real faults, but further experiments are necessary to confirm this hypothesis.

## V. RELATED WORK

A recent study on ANDROID test generation tools [9] examined 101 articles and found only few tools that generate ESPRESSO tests, while most tools only yield crash or exploration reports. For example, the popular MONKEY [34] random testing tool included in the ANDROID SDK only reports uncaught runtime exceptions during exploration; the same holds for many research prototypes such as DYNODROID [42], STOAT [43], APE [44], HUMANOID [45], TIMEMACHINE [46], Q-TESTING [47] or COMBODROID [48]. Some tools use a custom output format for tests, such as SAPIENZ [49] or MATE [11].

However, there are also tools that are able to generate ESPRESSO tests. Rohella et al. [50] briefly mention that their tool generates ESPRESSO tests, but do not provide any details on how this is implemented. COBWEB [51] builds on the ROBOLECTRIC [52] framework as an internal representation of tests and transforms tests into ESPRESSO format later, but unfortunately no description on how this is done is provided. RACERDROID [53] modifies the ESPRESSO framework to control event dispatching. It is unclear whether the ESPRESSO test cases generated by RACERDROID can be run outside the modified framework. DIFFDROID [28] extends MONKEY to automatically generate ESPRESSO tests, but aims at automatically finding cross-platform inconsistencies in ANDROID apps. Notably, none of the above tools presents an evaluation on the reliability of the generated ESPRESSO test cases.

The ESPRESSO test format is also used in record and replay tools such as Espresso Test Recorder (ETR) [26] or BARISTA [27]. However, these tools are integrated in ANDROID STUDIO and require manual interaction with the AUT, whereas the focus of our work is automated test generation.

There are also approaches to convert existing test cases into ESPRESSO format; for example, APPTESTMIGRATOR [54] migrates test cases in ESPRESSO format between apps of the same category (e.g., banking apps), and Coppola et al. [55] presented an approach for translating test scripts from visual-based tools into ESPRESSO test cases. However, both approaches focus on migrating existing manually written test cases rather than automated test generation.

Similar problems of creating locators as done by ESPRESSOMAKER also exist in other domains. For example, ROBULA+ [56] generates robust XPath locators for web testing. While the aim is similar, the technical challenges are different for XPath syntax and ESPRESSO.

Arcuschin et al. [9] studied the challenges of automatically synthesizing ESPRESSO test cases from sequences of interactions over ANDROID widgets, and concluded that the main challenge for properly generating reliable ESPRESSO test cases is the creation of non-failing ESPRESSO View Matchers. To tackle this issue, various heuristics have been proposed [9], [26]–[28], including checking the uniqueness of a view’s resource identifier, leveraging child and parent information, and properties such as class name or displayed text.

## VI. CONCLUSIONS AND FURTHER WORK

In this work, we presented a technique for generating reliable ESPRESSO test cases for ANDROID apps using novel algorithms for generating ESPRESSO View Matchers to concisely select ANDROID widgets, and for creating ESPRESSO View Assertions that serve for regression testing. This technique was implemented in ESPRESSOMAKER, an extension of the MATE testing tool for ANDROID test generation. ESPRESSOMAKER is open-source and publicly available [12], [13]. In summary, this article provides the following insights:

- The ESPRESSO framework can be used directly within an ANDROID test generation tool for collecting the available views and actions on any given screen.
- Concise and reliable ESPRESSO View Matchers for locating views in a UI can be generated automatically by iteratively combining *View Property Constraints*.
- ESPRESSO View Assertions for regression testing can be obtained by comparing the *View Properties* of the AUT before and after the execution of an action.

Our comprehensive empirical study on 1,035 ANDROID apps shows that ESPRESSOMAKER generates ESPRESSO tests that are significantly more reliable across different benchmarks, and that ESPRESSOMAKER generates ESPRESSO assertions that are statistically better at detecting faults than tests without assertions. For practical application, we present two assertion levels, *SemiFull* and *DiffOnly*, that significantly reduce the number of assertions, while still identifying a large portion of the non-equivalent mutants detected. In the spirit of transparency, reproducibility, and replicability, we provide a replication package [14] containing our source code, the scripts and the raw data collected during the empirical study.

As future work, we plan to continue our work on generating ESPRESSO assertions for regression testing. Specifically, we intend to explore the automatic generation of ESPRESSO assertions by dynamically detecting potential UI *invariants* in the apps under test [57]. Additionally, we plan to analyze and improve the readability of the generated ESPRESSO tests. We have taken initial steps in this direction by using Delta Debugging to minimize ESPRESSO View Matchers, but we plan to explore other techniques such as Large Language Models to generate more readable test cases [58].

## ACKNOWLEDGMENTS

This work is supported by DFG project FR2955/4-1 “STUNT: Improving Software Testing Using Novelty”, UBACYT-2020 20020190100233BA and PICT-2019-01793.

## REFERENCES

- [1] “Espresso — Android Developers,” <https://developer.android.com/training/testing/espresso>, (Accessed on 10/24/2023).
- [2] L. Cruz, R. Abreu, and D. Lo, “To the attention of mobile software developers: guess what, test your app!” *Empirical Software Engineering*, vol. 24, no. 4, pp. 2438–2468, 2019.
- [3] M. Linares-Vásquez, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk, “How do developers test android applications?” in *2017 IEEE International Conference on Software Maintenance and Evolution (IC-SME)*. IEEE, 2017, pp. 613–622.
- [4] S. R. Choudhary, A. Gorla, and A. Orso, “Automated Test Input Generation for Android: Are We There Yet? (E),” in *ASE*. IEEE Computer Society, 2015, pp. 429–440.
- [5] H. Zheng, D. Li, B. Liang, X. Zeng, W. Zheng, Y. Deng, W. Lam, W. Yang, and T. Xie, “Automated Test Input Generation for Android: Towards Getting There in an Industrial Case,” in *ICSE-SEIP*. IEEE Computer Society, 2017, pp. 253–262.
- [6] W. Wang, D. Li, W. Yang, Y. Cao, Z. Zhang, Y. Deng, and T. Xie, “An empirical study of Android test generation tools in industrial cases,” in *ASE*. ACM, 2018, pp. 738–748.
- [7] P. Kong, L. Li, J. Gao, K. Liu, T. F. Bissyandé, and J. Klein, “Automated Testing of Android Apps: A Systematic Literature Review,” *IEEE Trans. Reliab.*, vol. 68, no. 1, pp. 45–66, 2019.
- [8] T. Su, J. Wang, and Z. Su, “Benchmarking automated GUI testing for Android against real-world bugs,” in *ESEC/SIGSOFT FSE*. ACM, 2021, pp. 119–130.
- [9] I. A. Moreno, J. P. Galeotti, C. Ciccaroni, and J. M. Rojas, “On the feasibility and challenges of synthesizing executable Espresso tests,” in *AST@ICSE*. ACM/IEEE, 2022, pp. 92–102.
- [10] “AccessibilityService — Android Developers,” <https://developer.android.com/reference/android/accessibilityservice/AccessibilityService>, (Accessed on 10/24/2023).
- [11] M. M. Eler, J. M. Rojas, Y. Ge, and G. Fraser, “Automated Accessibility Testing of Mobile Apps,” in *ICST*. IEEE Computer Society, 2018.
- [12] “EspressoMaker’s open-source implementation (MATE extension),” <https://github.com/FlyingPumba/EspressoMaker-mate>, (Accessed on 10/24/2023).
- [13] “EspressoMaker’s open-source implementation (MATE server extension),” <https://github.com/FlyingPumba/EspressoMaker-mate-server>, (Accessed on 10/24/2023).
- [14] “Replication package,” <https://zenodo.org/records/10038308>, (Accessed on 10/24/2023).
- [15] “JUnit framework for Java unit testing,” <https://junit.org>, (Accessed on 10/24/2023).
- [16] “Build instrumented tests — Android Developers,” <https://developer.android.com/training/testing/instrumented-tests>, (Accessed on 10/24/2023).
- [17] “Instrumentation — Android Developers,” <https://developer.android.com/reference/android/app/Instrumentation.html>, (Accessed on 10/24/2023).
- [18] “Appium: Mobile App Automation Made Awesome.” <http://appium.io/>, (Accessed on 10/24/2023).
- [19] “calabash/calabash-android: Automated Functional testing for Android using cucumber,” <https://github.com/calabash/calabash-android>, (Accessed on 10/24/2023).
- [20] “monkeyrunner — Android Developers,” <https://developer.android.com/studio/test/monkeyrunner>, (Accessed on 10/24/2023).
- [21] “RobotiumTech/robotium: Android UI Testing,” <https://github.com/RobotiumTech/robotium>, (Accessed on 10/24/2023).
- [22] “Write automated tests with UI Automator — Android Developers,” <https://developer.android.com/training/testing/other-components/ui-automator>, (Accessed on 10/24/2023).
- [23] “AndroidX Test Releases — Android Developers,” <https://developer.android.com/jetpack/androidx/releases/test>, (Accessed on 10/24/2023).
- [24] L. Sell, M. Auer, C. Frädriich, M. Gruber, P. Werli, and G. Fraser, “An Empirical Evaluation of Search Algorithms for App Testing,” in *ICTSS*, ser. LNCS, vol. 11812. Springer, 2019.
- [25] “android-test/RootsOracle.java at master — android/android-test · GitHub,” <https://github.com/android/android-test/blob/master/espresso/core/java/androidx/test/espresso/base/RootsOracle.java>, (Accessed on 10/24/2023).
- [26] S. Negara, N. Esfahani, and R. P. L. Buse, “Practical Android test recording with espresso test recorder,” in *ICSE (SEIP)*. IEEE / ACM, 2019, pp. 193–202.
- [27] M. Fazzini, E. N. de A. Freitas, S. R. Choudhary, and A. Orso, “Bariستا: A Technique for Recording, Encoding, and Running Platform Independent Android Tests,” in *ICST*. IEEE Computer Society, 2017, pp. 149–160.
- [28] M. Fazzini and A. Orso, “Automated cross-platform inconsistency detection for mobile apps,” in *ASE*. IEEE Computer Society, 2017, pp. 308–318.
- [29] A. Zeller and R. Hildebrandt, “Simplifying and Isolating Failure-Inducing Input,” *IEEE Trans. Software Eng.*, vol. 28, pp. 183–200, 2002.
- [30] T. Xie, “Augmenting Automatically Generated Unit-Test Suites with Regression Oracle Checking,” in *European Conference on Object-Oriented Programming*, 2006.
- [31] G. Fraser and A. Zeller, “Mutation-Driven Generation of Unit Tests and Oracles,” *IEEE Transactions on Software Engineering*, vol. 38, pp. 278–292, 2010.
- [32] “Android Interface Definition Language (AIDL) — Android Developers,” <https://developer.android.com/guide/components/aidl>, (Accessed on 10/24/2023).
- [33] K. Allix, T. F. Bissyandé, J. Klein, and Y. L. Traon, “AndroZoo: Collecting Millions of Android Apps for the Research Community,” *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pp. 468–471, 2016.
- [34] “UI/Application Exerciser Monkey — Android Developers,” <https://developer.android.com/studio/test/other-testing-tools/monkey>, (Accessed on 10/24/2023).
- [35] C. Escobar-Velásquez, D. Riveros, and M. Linares-Vásquez, “MutAPK 2.0: a tool for reducing mutation testing effort of Android apps,” in *ESEC/SIGSOFT FSE*. ACM, 2020, pp. 1611–1615.
- [36] “Yuan-W/muDroid: Mutation Testing tool for Android Integration Testing,” <https://github.com/Yuan-W/muDroid>, (Accessed on 10/24/2023).
- [37] J. Liu, X. Xiao, L. Xu, L. Dou, and A. Podgurski, “DroidMutator: an effective mutation analysis tool for Android applications,” in *ICSE (Companion Volume)*. ACM, 2020, pp. 77–80.
- [38] A. Arcuri and L. C. Briand, “A Hitchhiker’s guide to statistical tests for assessing randomized algorithms in software engineering,” *Softw. Test., Verif. Reliab.*, vol. 24, no. 3, pp. 219–250, 2014.
- [39] A. Arcuri and G. Fraser, “Parameter tuning or default values? An empirical investigation in search-based software engineering,” *Empirical Software Engineering*, vol. 18, no. 3, pp. 594–623, 2013.
- [40] Y. Jia and M. Harman, “An Analysis and Survey of the Development of Mutation Testing,” *IEEE Trans. Software Eng.*, vol. 37, no. 5, pp. 649–678, 2011.
- [41] M. L. Vásquez, G. Bavota, M. Tufano, K. Moran, M. D. Penta, C. Vendome, C. Bernal-Cárdenas, and D. Poshyvanyk, “Enabling mutation testing for Android apps,” in *ESEC/SIGSOFT FSE*. ACM, 2017, pp. 233–244.
- [42] A. Machiry, R. Tahiliani, and M. Naik, “Dynodroid: an input generation system for Android apps,” in *ESEC/SIGSOFT FSE*. ACM, 2013, pp. 224–234.
- [43] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, “Guided, stochastic model-based GUI testing of Android apps,” in *ESEC/SIGSOFT FSE*. ACM, 2017.
- [44] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, and Z. Su, “Practical GUI testing of Android applications via model abstraction and refinement,” in *ICSE*. IEEE / ACM, 2019, pp. 269–280.
- [45] Y. Li, Z. Yang, Y. Guo, and X. Chen, “Humanoid: A Deep Learning-Based Approach to Automated Black-box Android App Testing,” in *ASE*. IEEE, 2019, pp. 1070–1073.
- [46] Z. Dong, M. Böhme, L. Cojocaru, and A. Roychoudhury, “Time-travel testing of Android apps,” in *ICSE*. ACM, 2020, pp. 481–492.
- [47] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, “Reinforcement learning based curiosity-driven testing of Android applications,” in *ISSTA*. ACM, 2020, pp. 153–164.
- [48] J. Wang, Y. Jiang, C. Xu, C. Cao, X. Ma, and J. Lu, “ComboDroid: generating high-quality test inputs for Android apps via use case combinations,” in *ICSE*. ACM, 2020, pp. 469–480.
- [49] K. Mao, M. Harman, and Y. Jia, “Sapienz: multi-objective automated testing for Android applications,” in *ISSTA*. ACM, 2016.
- [50] A. Rohella and S. Takada, “Testing Android Applications Using Multi-Objective Evolutionary Algorithms with a Stopping Criteria,” in *SEKE*.

KSI Research Inc. and Knowledge Systems Institute Graduate School, 2018, pp. 308–307.

- [51] R. Jabbarvand, J. Lin, and S. Malek, “Search-Based Energy Testing of Android,” in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 1119–1130.
- [52] “Robolectric,” <https://robolectric.org/>, (Accessed on 10/24/2023).
- [53] H. Tang, G. Wu, J. Wei, and H. Zhong, “Generating test cases to expose concurrency bugs in Android applications,” in *ASE*. ACM, 2016, pp. 648–653.
- [54] F. Behrang and A. Orso, “Test Migration Between Mobile Apps with Similar Functionality,” in *ASE*. IEEE, 2019, pp. 54–65.
- [55] R. Coppola, L. Ardito, M. Torchiano, and E. Alégroth, “Translation from Visual to Layout-based Android Test Cases: a Proof of Concept,” in *ICST Workshops*. IEEE, 2020, pp. 74–83.
- [56] M. Leotta, A. Stocco, F. Ricca, and P. Tonella, “Robula+: an algorithm for generating robust XPath locators for web testing,” *J. Softw. Evol. Process.*, vol. 28, no. 3, pp. 177–204, 2016.
- [57] J. C. Alonso, S. Segura, and A. Ruiz-Cortés, “AGORA: Automated Generation of Test Oracles for REST APIs,” in *ISSTA*. ACM, 2023, pp. 1018–1030.
- [58] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J. Lou, and W. Chen, “CodeT: Code Generation with Generated Tests,” in *ICLR*. OpenReview.net, 2023.