Automated Translation of Android Context-Dependent Gestures to Visual GUI Test Instructions

Riccardo Coppola
riccardo.coppola@polito.it
Politecnico di Torino
Turin, Italy

Luca Ardito
luca.ardito@polito.it
Politecnico di Torino
Turin, Italy

Marco Torchiano
marco.torchiano@polito.it
Politecnico di Torino
Turin, Italy

ABSTRACT

Layout-based (2nd generation) and Visual (3rd generation) GUI testing are two very common approaches for mobile application testing. The two techniques expose complementary advantages and drawbacks, and the literature on GUI Testing has highlighted the benefits of an approach based on a translation from one generation to the other.

The objective of this work is to provide an improvement to our prototype tool, TOGGLE, designed to translate 2nd generation test suites, written with the Espresso framework, to 3rd generation ones that can be run by the EyeAutomate and Sikuli tool.

We extended TOGGLE by adding (1) support for context-based gestures, performed through the *scrollTo* and *onData* commands, and (2) support for the combination of *Layout-based* locators with logical operators.

We evaluated the new version of the tool on five different experimental subjects. For each of the applications, 30 test cases were developed and automatically translated with TOGGLE+.

We observed an increase of 68% of translatable test cases when transitioning from the previous prototype to the current version of the tool. The generated *Visual* test cases also proved to have high robustness, with flakiness of just 2% (i.e., 98% correct executions).

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**; **Maintaining software**.

KEYWORDS

GUI Testing, Android Testing

ACM Reference Format:
Riccardo Coppola, Luca Ardito, and Marco Torchiano. 2021. Automated Translation of Android Context-Dependent Gestures to Visual GUI Test Instructions. In *Proceedings of the 12th International Workshop on Automating TEST Case Design, Selection, and Evaluation (A-TEST '21), August 23–24, 2021, Athens, Greece.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3472672.3473954

A-TEST '21, August 23–24, 2021, Athens, Greece

ACM ISBN 978-1-4503-8623-4/21/08…$15.00
https://doi.org/10.1145/3472672.3473954

1 INTRODUCTION

Mobile devices and mobile apps are highly pervasive in our everyday life, supporting a wide range of activities and providing complex user experiences and interactions. Such complexity in user interaction, paired with the variety of the Graphical User Interfaces provided (GUI), drive a need for thorough testing and verification in this software domain.

Many tools are available in technical and academic literature to allow testing replicating the users' operations with Android GUIs. These tools can be based on the identification and verification of GUI components through exact screen coordinates (1st generation or coordinate-based tools), text-based properties of the visualized GUI components (2nd generation or *Layout-based* tools) or – as in more recent research efforts – through the application of image recognition techniques (3rd generation generation or *Visual* tools). However, while low-level unit testing not involving the GUI is a widespread practice, these higher-level testing techniques are not extensively adopted by practitioners, and manual execution of test sequences is preferred [8].

1st generation techniques have generally been abandoned because of the intrinsic inaccuracy of coordinate-based locators. Research into 2nd generation and 3rd generation GUI testing techniques, instead, has shown that they still expose complementary benefits and drawbacks. A combined application of them (through translation-based test creation) can mitigate the most crucial deterrents to the adoption of GUI testing, i.e. the high fragility of test cases during the expected evolution of the System Under Test (SUT) [1], and the problematic adaptability of test cases to different hardware and software configurations.

In our previous work, we have conceptualized and developed a prototype framework, TOGGLE, for the translation of Android *Layout-based* tests written with Espresso to two state-of-the-art *Visual* testing tools, EyeAutomate and Sikuli[5]. The tool featured an initial set of interactions akin to those implemented in state-of-the-art tools to translate tests for web applications [11]. Still, it provided partial coverage of all the possible actions that can be executed by the Espresso framework or by other similar *Layout-based* GUI testing tools for Android applications. With this work, we provide the following contributions:

- We extend the current state-of-the-art in the field of Android GUI testing by defining a methodology for representing context-dependent gestures in *Visual* GUI test suites;
- We implement these translation mechanisms in a revised version of our tool, TOGGLE+;
- We experimentally evaluate the ability of the new tool in translating Android-specific gestures to *Visual* test cases.

The remainder of the manuscript is organized as follows: Section 2 provides background information about Android testing tools, their drawbacks, and the benefits of a translation-based approach; Section 3 includes information about the TOGGLE tool; Section 4 and describes the improvements that we applied to the framework; Section 5 describes the experimental evaluation of the tool and reports its results; Section 6 discusses the potential limitations of the present work; Section 7 concludes the paper by summarizing the findings and providing hints for future research directions.

2 BACKGROUND

This section describes the way Android GUIs are organized and the available technologies for testing Android apps, along with their main drawbacks. We also discuss the motivation and the benefits of a translation-based approach combining the characteristics of *Layout-based* and *Visual* testing.

2.1 Android Native App Structure

Android applications can be divided into three categories:

- *Native*, when developed for Android devices specifically using constructs specific of the development platform;
- *Web-based*, when developed as canonical web applications and then optimized for the fruition on mobile devices (e.g., *progressive* web apps);
- *Hybrid*, when developed using multi-platform platforms (e.g., Flutter or React Native).

The behaviour for each GUI screen of Android apps is defined by components named *Activities*. Each activity populates the screen with widgets according to the arrangement defined in XML *layout files*. Layout files define a standard set of properties for each component, many of which are text-based and can be used as locators or oracles for *Layout-based* testing techniques (e.g., ids, contained text, content descriptions, *hints* in text input boxes). Multiple layouts can be defined for the same activity to provide a different arrangement or appearance of the widgets based on orientation, resolution and other characteristics of the device where the app is launched. Specific layout components (e.g., *AdapterViews* or *GridViews*) are typically populated dynamically at runtime.

2.2 Tools for Android GUI Testing

Layout-based testing techniques for Android apps are typically used to test Native applications since they are engineered to recognize the elements of the GUI by using domain-specific layout properties.

By adopting a classification scheme provided by Linares-Vasquez et al. [10], we define *Automation APIs* the tools providing means of writing JUnit-like sequences of interactions and assertions against the widget of the SUT. The most widespread tools of this category are the built-in Espresso, and UIAutomator frameworks [14], and the multi-platform Appium framework [12].

More recent *Model-based* tools, like SAPIENZ, [2], and Droidbot [9], leverage the definition of GUI components to create models of the GUI that can be exploited to define and execute test cases automatically.

Visual testing tools have been rarely defined specifically for mobile applications in the literature. These tools are, however, inherently platform-agnostic. Hence they can be applied to any SUT emulated on a desktop device where the image recognition engine is launched. Tools like Sikuli [13] and EyeAutomate [3] have been used for research on mobile testing. The main limitation of *Visual* testing tools when applied to Android SUTS is the lack of built-in support for Android-specific commands (e.g., gestures) that have to be emulated through basic mouse and keyboard operations on the virtual devices.

2.3 Limitations of Android GUI Testing

One of the most critical issues impacting Android GUI testing is *Fragmentation*, i.e., the wide variety of devices, screen sizes, and versions of the Android operating system with which an application must provide compatibility. This peculiarity of the mobile domain mandates the need for testing each app on all such configurations. Test suites, especially *Visual* ones, require a significant effort to be adapted to the various configurations considered [7].

Another critical weakness of mobile GUI testing is *Fragility*. A test case is said *fragile* when it fails because of changes in the locators or oracles it uses (layout properties for *Layout-based* tests and widget captures for *Visual* tests).

Fragility can frequently manifest during the lifespan of a mobile application, given the fast-paced evolution of the hardware and software configurations it must be released on. The adaptation of test cases may sum up to a significant fraction of the total maintenance cost of the whole software project [6].

2.4 Benefits of a Translation-Based Approach

Works in the literature have testified that a combination of *Layout-based* and *Visual* GUI testing approaches can be used to mitigate the respective drawbacks of the two techniques used alone [1]. The most common methodology adopted to combine the two techniques is a translation-based approach to automatically convert locators and oracles based on layout properties to screen captures that image recognition algorithms can employ.

The benefits of the translation-based approach are manifold. First, *Layout-based* tests and *Visual* tests, even if exercising the same user scenarios against the SUT, can detect complementary types of mutants: respectively, errors in the properties and composition of the GUI, and errors in the pictorial rendering of the interface. The automated creation of *Visual* counterparts of *Layout-based* tests can therefore enhance the defect-finding capabilities of an existing test suite at a minimal cost.

As well, *Layout-based* and *Visual* tests are fragile to complementary modifications in the SUT. The translation-based approach can reduce the test maintenance costs by automatically analyzing the failing locators and repairing them based on the still valid counterparts. For instance, if a *Visual* locator can no longer be found because the appearance of a widget is changed in a new release of the app, it can be re-generated automatically by using the unchanged associated *Layout-based* properties.

Finally, *Visual* scripts expose very limited robustness to variations in the screen resolution or aspect ratio, whilst *Layout-based* ones generally have higher portability to varying configurations. By generating *Visual* scripts through reuse of *Layout-based* tests, it is possible to port them to different configurations automatically.

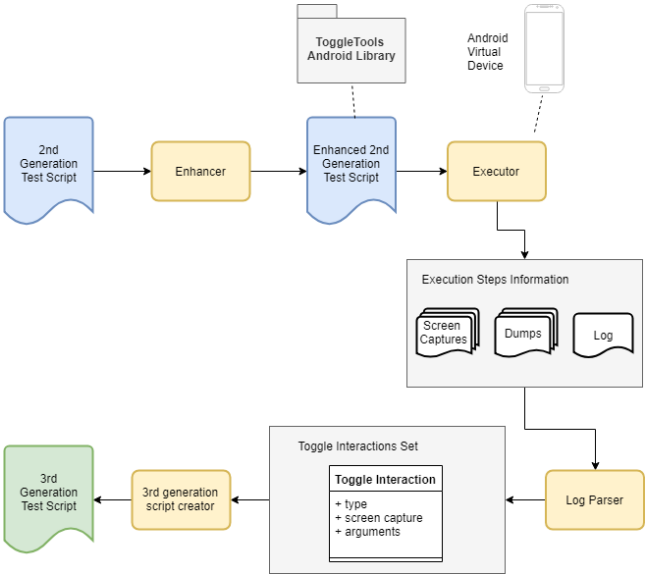

Figure 1: Architecture of TOGGLE

Table 1: Commands covered by the TOGGLE Script Creator

Espresso command	Android-specific	Required *Visual* instructions
Click	No	1
Double Click	No	2
Long click	No	3
Press Back	Yes	1
PressKey	No	1
PressMenuKey	Yes	1
CloseSoftKeyboard	Yes	1
Swipe[Up/Left/Down/Right]	Yes	4
ClearText	No	2
TypeIntoFocusedView	Yes	1
TypeText	No	2
ReplaceText	No	3

```
@Test
public void testTest() {

    onView(withId(R.id.fab_expand_menu_button)).perform(click());

    onView(withText("Text note")).perform(click());

}
```

(a) Original test case

```
testAddNote, testAddNote1, id, fab_expand_menu_button, click,
testAddNote, testAddNote2, text, Text note, click
testAddNote, testAddNote3, id, detail_title, typetext, Text
testAddNote, testAddNote4, content-desc, drawer open, click,
testAddNote, testAddNote5, content-desc, drawer open, click,
testAddNote, testAddNote6, id, settings_view, click,
```

(b) Log excerpt

```
<node bounds="[0,1269][1080,1794]" visible-to-
    user="true" selected="false" password="false" long-
    clickable="false" scrollable="false" focused="false"
    focusable="false" enabled="true" clickable="false"
    checked="false" checkable="false" content-desc=""
    package="it.feio.android.omninotes.foss"
    class="android.view.ViewGroup" resource-
    id="it.feio.android.omninotes.foss:id/snackbar_placeholder"
    text="" index="2"/>
<node bounds="[626,957][1059,1773]" visible-to-
user="true" selected="false" password="false" long-
clickable="false" scrollable="false" focused="false"
focusable="false" enabled="true" clickable="false"
checked="false" checkable="false" content-desc=""
package="it.feio.android.omninotes.foss"
class="android.view.ViewGroup" resource-
id="it.feio.android.omninotes.foss:id/fab" text=""
index="3">
    <node bounds="[865,1579][1059,1773]" visible-to-
        user="true" selected="false" password="false" long-
        clickable="true" scrollable="false" focused="false"
        focusable="true" enabled="true" clickable="true"
        checked="false" checkable="false" content-desc=""
        package="it.feio.android.omninotes.foss"
        class="android.widget.ImageButton" resource-
        id="it.feio.android.omninotes.foss:id/fab_expand_menu_button"
        text="" index="6" NAF="true"/>
```

(c) Screen hierarchy (*dump*) with highlighted *Layout-based* locator

(d) Visual locator for the interacted widget

Figure 2: Translation steps

While our previous work and related ones have empirically proved these benefits, a limitation of current translation-based approaches is the absence of a translation of complex gestures and assertions that are specific to the Android platform. Those complex actions cannot be directly translated to equivalent *Visual* interactions performed through mouse and keyboard commands. The work described in this paper aims at overcoming such limitations.

3 THE TOGGLE TOOL

The TOGGLE tool, which was initially described and evaluated in our previous works [4][5], is a tool to perform automated translation of *Layout-based* test suites written with the Espresso framework to *Visual* test suites in the syntax of Sikuli and EyeAutomate. We originally chose Espresso as the starting *Layout-based* tool since – based on our previous analyses – it is the most widespread GUI testing tools used in open-source Android repositories [6].

The tool receives as input a *Layout-based* test class, executes it against an emulated Android Virtual Device and generates cropped screen captures of the GUI that can be used as oracles and locators for *Visual* test suites.

The architecture of TOGGLE, shown in the diagram in fig. 1 is based on four main components:

- **Enhancer**: it parses the *Layout-based* script passed as input (fig. 2a) and injects functions of the TOGGLE library, that are required to capture the required information for the translation when the test is executed.
  Specifically, the TOGGLE library features methods to obtain, at each interaction with the GUI, the fullscreen capture and the *dump* file, an XML representation of the current screen including all widget properties.
  The enhancer also injects calls to a logging method to define the trace of interactions that have to be translated. In the original version of TOGGLE, the Enhancer module can recognize only Espresso assertions starting with the *onView* method. The tool can interact only with individual widgets and not with dynamically populated widget lists (i.e., methods starting with the *onData* assertion).
- **Executor**: it executes the enhanced test script against the selected Android Virtual Device (AVD). The executor launches the device, installs the Application Under Test (AUT), and

runs the test so that – with the previously injected calls – screen captures, properties, and interaction traces are produced and saved on the device memory.
- **Log Parser**: it parses the log (fig. 2b) generated during the execution of the test. For each interaction, it retrieves:
  – the type of interaction;
  – the widget on which it has been performed and,
  – its parameters (e.g., a string for a text input).
  It then uses the dump file (Figure. 2c) to retrieve the bounding box of the widget to cut it from the related screen capture, and generates the *Visual* locator (fig. 2d). For each interaction, it creates a ToggleInteraction object.
- **Visual script creator**: it receives the list of ToggleInteraction objects and translates each interaction to the syntax of the desired *Visual* testing tool (e.g., Python scripts for SikuliX, and plain text files for EyeAutomate). A 1-to-1 mapping from *Layout-based* to *Visual* instructions is generally not possible since the latter has to be performed through generic mouse and keyboards interaction. Table 1 show the commands supported by the original version of the tool.

4 IMPROVEMENT OF THE TOGGLE TOOL

Starting from the original prototype of TOGGLE, we developed extensions of the tool to increase the number of operations that can be effectively translated automatically to *Visual* test scripts. The source code of the TOGGLE+ tool is available as an open-source repository[1].

4.1 Scroll Operations

We added support for the *scroll* interaction with dynamically populated containers of widgets in the AUT. It is worth mentioning that for Android AUTs, the scroll interaction is different from the *swipe* interaction, which can be performed on statically defined views as well.

The scroll interaction can be performed through two different commands with the Espresso framework, *onData* and *scrollTo*. The translation of both commands, however, follows a typical sequence of steps that is now detailed.

First, a sequence of instructions has to be added to the test code by the Enhancer when parsing the *Layout-based* test scripts and identifying scroll commands to generate the following elements when the Executor runs the test:

(1) A reference to the container of the View where the scroll is executed. The way the object is referenced depends on the specific Espresso command, as detailed in subsections 4.1.1 and 4.1.2;
(2) A screen capture of the GUI appearance before the scrolling operation (i.e., starting situation), as well as the corresponding XML dump file;
(3) Starting coordinates (x_s, y_s) of the scroll interaction;
(4) Ending coordinates (x_e, y_e) of the scroll interaction;
(5) A screen capture of the GUI appearance after the scrolling operation (i.e., ending situation), as well as the corresponding XML dump file;
(6) A log line reporting the information described in the previous points.

After the test is executed and the information above is collected, the Log Parser module of the TOGGLE+ architecture handles creating the ToggleInteraction object representing the scroll interaction. The main operations performed by the Log Parser module are the following:

(1) The image locator to be used in the *Visual* test script is extracted by retrieving the boundaries of the elements in the XML dump and cutting a sub-image of the screen capture of the starting situation;
(2) The direction and offset (x_o, y_o) of the scroll is computed, based on starting and ending coordinates. The computed offset allows defining one of two specializations of the ToggleInteraction object, VerticalScroll or HorizontalScroll.

The *Visual* Script Creator, finally, utilizes the information in the ToggleInteraction object to generate commands for the selected *Visual* testing tool syntax. Within a *Visual* test script, a scroll interaction is obtained by a pressure on a specific point on the screen, a movement along the scroll direction and, finally, the release of the pressure. This operation has to be repeated in a loop until the destination View of the scroll is displayed on the screen, if the offset to reach the ending situation is bigger than the size of the View. The flow chart representing a scroll operation in a *Visual* test script is represented in figure 3.

The number of movements performed in the loop depends on the scroll offset and the size of the scrollable View in the direction of the scroll, i.e. the *scrollStep*. The number of movements can hence be computed as $floor(x_o/scrollStep)$ in case of horizontal scroll, or $floor(y_o/scrollStep)$ in case of vertical scroll.

If the offset to scroll is not an exact multiple of the scrollStep, the remainder is computed as $x_o \ mod \ scrollStep$ in case of the horizontal scroll, or $y_o \ mod \ scrollStep$ in case of the vertical scroll.

In the recreation of the scroll movement, we had to empirically tune the timing between each couple of subsequent interactions to perform with the SUT. The refresh time required by the GUI and the speed of the movements of the cursor can impact the precision of the operations performed in terms of ending coordinates reached after the scroll. We added a sleep time of 0.1 seconds between the start of the pressure on the *Visual* locator and the actual movement and a sleep time of 1 second after the end of each movement and the release. Additionally, since performing too long movements could lead to losing focus of the current View in the emulated device, we divided each individual movement (of *scrollStep* length) into three segments of equal length, without releasing the pressure on the View. A sleep time of 0.25s is inserted between each segment. The evaluation of sleep times was performed by gradually reducing the injected time until flakiness was encountered in test suite execution.

4.1.1 onData. The *onData* method is a type of ViewAssertion, designed to perform an interaction on an *AdapterView* (i.e., an Android View whose purpose is to display a group of child elements contained in an adapter), finding among its child the one that matches the specified conditions, scrolling to it, and then optionally executing one interaction on it. The method is the principal way of interacting with dynamically-populated widget containers, for

[1] https://git-softeng.polito.it/d023270/toggle-v2

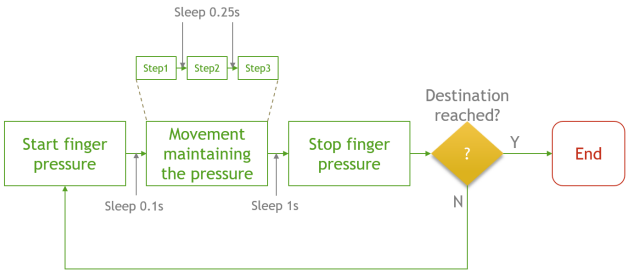

Figure 3: Flowchart for a scroll operation in a *Visual* test script

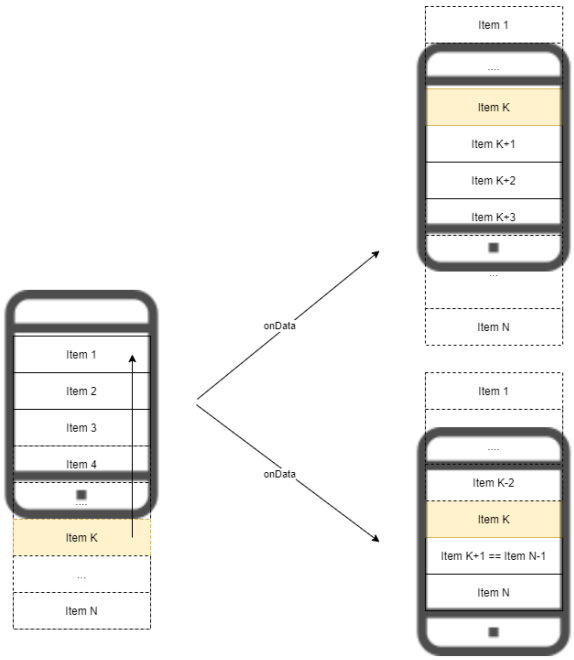

Figure 4: Espresso.onData() scrolling semantics.

```
onData(inAdapterView(withId(R.id.
    ↪ settings_category_list)).atPosition(12).check
    ↪ (matches(isDisplayed));
```

Figure 5: Example of scroll command with onData

which it is not possible to access the individual elements through the *onView* method, previously implemented in TOGGLE.

As displayed in figure 4, the *onData* method scrolls the widget container until the searched element is brought to the first available position (the upmost one in case of vertically scrollable container, and the leftmost one in case of horizontally scrollable container), or it is not possible to scroll the View any longer.

Figure 5 represents the way a scroll operation is defined in Espresso with the usage of the *onData* method: the method receives the id of the Adapter as a parameter; it is then possible to

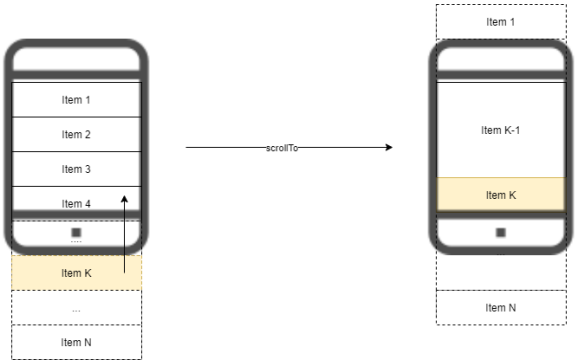

Figure 6: ViewActions.scrollTo() scrolling semantics.

```
onView(withId(R.id.view_id))
    .perform(scrollTo());
```

Figure 7: Example of scroll command with scrollTo

define a position to which the scroll has to be performed (alternatively, the specific element can be identified based on its textual properties); finally, the operation to perform on the View once it is reached (in the example, the verification of its presence) is specified. The reference point for the start of the scroll operation is the first element in the ViewAdapter.

The Enhancer module identifies the presence of the *inAdapterView* keyword to identify the View where the scroll has to be performed and computes the coordinates and the offsets by using the getter methods of the adapterView. The Espresso ViewAction performed once the element is found and made visible in the Adapter View is then logged as a separate *Visual* operation to be translated to the destination scripting syntax.

4.1.2 scrollTo. The *scrollTo* method is a ViewAction. Thus it is used as the conclusion of an Espresso command after a View is identified (see figure 8).

Such ViewAction applies only to instances of the ScrollView and HorizontalScrollView classes and scrolls to an existing layout element until it is displayed (as shown in figure 6). The element to which the scroll has to be performed is identified through text-based properties. No movement is performed on the View if the element is already visible on the screen.

When scrollTo operations are found, the Enhancer injects instructions to log the first element in the ScrollView and computes the offset to reach the ending point after the searched element is made visible.

4.2 AllOf - AnyOf View Assertions

The *allOf()* and *anyOf()* Espresso commands express a logical relationship between different ViewMatchers that can be used to identify a View in the currently inflated layout hierarchy. Their purpose is to express, respectively, a logical *and* or a logical *or* condition between the results of two ViewMatchers.

```
onView(allOf(withId(R.id.view), withText("Hello!")));

onView(anyOf(withId(R.id.view), withText("Hello!")));
```

Figure 8: Examples of allOf - anyOf commands

Table 2: Considered experimental subjects.

Application	PlayStore downloads	GitHub stars	GitHub commits	Lines of code	Last update
Budget Watch	29	70	477	9860	27/01/2021
Contact Book	-	7	26	2980	02/08/2019
PDF Converter	952	607	639	19110	23/03/2021
Simple Calendar	6805	2397	4350	28762	22/03/2021
Stoic Reading	-	14	235	31836	05/02/2021

Table 3: Distribution of Espresso commands and assertions per experimental subject

Interaction	Budget Watch	Contact Book	PDF CONVERTER	Simple Calendar	Stoic Reading
allOf	1	1	34	8	0
anyOf	2	0	0	0	0
atPosition	21	9	0	0	0
click	37	30	7	30	20
inAdapterView	21	9	0	0	0
instanceOf	0	0	34	0	0
isDisplayed	17	24	11	12	9
matches	23	24	11	12	9
onData	21	9	0	0	0
onView	44	46	37	63	52
scrollTo	3	0	19	12	23
typeText	2	1	0	9	0
withContentDesc.	2	0	4	5	0
withId	55	53	33	62	50
withText	9	3	0	4	2
All	258	209	190	217	165

The support for these instructions, previously missing in the original prototype of the TOGGLE tool, has been achieved by slightly modifying the ToggleInteraction class and defining an array of *SearchType* and *SearchKeywords* instead of an individual value and a boolean parameter in the ToggleInteraction used to define whether the logical *and* or *or* applies.

The Enhancer method, when *allOf()* or *anyOf()* calls are encountered, has been modified to support multiple keywords instead of single one per log line.

5 EVALUATION

This section describes the experimental evaluation conducted on TOGGLE+, the adopted procedure and its results.

5.1 Experimental Subjects and Environment

For the evaluation of the applicability of the tool in translating Espresso *Layout-based* commands, we needed a set of Android projects containing Espresso test suites. We initially mined GitHub repositories containing code written with the Espresso framework; however, the number and size of test suites did not prove to be sufficient for our purposes in any of the projects. Therefore, we selected five applications on which we developed Espresso test suites. The applications were selected from the F-Droid open-source app repository[2] based on the following criteria: (i) the application had to be native; (ii) the code of the application had to be hosted on GitHub; (iii) the application had to be updated at least once in last two years; (iv) the application had to have a sufficient complexity (at least two thousand LOCs). The set of selected applications, along with information about their size and diffusion among end-users and developers' community, is reported in table 2.

For each experimental subject, one author of the paper developed a suite of 30 test cases. The author was not involved in developing the TOGGLE+ modules and was only given the list of commands supported by the tool for the definition of test cases. The distribution of the Espresso commands we used in the test suites is reported in table 3. Over the 150 tests, 45% of them execute at least one scroll interaction (with a top 80% for Budget Watch).

The hardware infrastructure used to execute all test cases is an Acer Aspire A715-71G with an Intel(R) Core(TM) i7-7700HQ CPU at the frequency of 2.80GHz, 16GB of RAM, and running Windows 10 Home as the operating system. All Espresso test cases have been executed on a Nexus 5 API 28 (Android 9) with disabled animations.

All 3^{rd} generation test executions have been performed on a black background to avoid any disturbances to the image recognition algorithms and executed with the SikuliX tool.

5.2 Research Questions and Procedure

The evaluation of the TOGGLE+ tool entailed answering two research questions, detailed in the following along with the adopted procedure.

- **RQ1 - Translation Precision**: What is the efficiency of the TOGGLE+ tool in translating test cases?

To answer RQ1, we measured the ratio between the test cases that were correctly translated by the TOGGLE+ tool (i.e., those for which a valid *Visual* test script was created by applying the tool) and the total number of test cases to translate (i.e., 30 per each experimental subject).

- **RQ2 - *Visual* Script Success Rate**: What is the success rate of the *Visual* test scripts generated through translation?

To answer RQ2, we computed the Success Rate metric, defined as the ratio between the number of completely executed test cases and the total number of executions performed.

This evaluation is necessary since evidence reports that *Visual* test script has high flakiness. *Visual* test scripts can have aleatory results. They can fail because of imprecision of the image recognition algorithm, timing issues with the emulated devices, and other issues implying that the *Visual* locators are not properly loaded at the time of image search.

To evaluate the inherent flakiness of *Visual* scripts, we executed each successfully translated test case on the emulated device ten times.

- **RQ3 - *Visual* Interaction Success Rate**: What is the success rate of the individual *Visual* interactions generated through translation?

To answer RQ3, the ratio of successfully executed interactions, i.e. the ratio of interactions executed correctly by the *Visual* test

[2] https://www.f-droid.org/

Table 4: Translation Precision of state of the art and re-engineered TOGGLE tool

Application	State-of-the-art Translatable tests	State-of-the-art Ratio	TOGGLE+ Translatable tests	TOGGLE+ Ratio
Budget Watch	5	17%	30	100%
Contact Book	22	73%	30	100%
PDF Converter	0	0%	30	100%
Simple Calendar	11	37%	30	100%
Stoic Reading	10	33%	30	100%
Total	48	32%	150	100%

Table 5: Success rate of *Visual* test execution per experimental subject

Application	Successful executions	Failing executions	Success Rate
Budget Watch	292	8	97.3%
Contact Book	298	2	99.3%
PDF Converter	282	18	94%
Simple Calendar	300	0	100%
Stoic Reading	298	2	99.3%
Total	1470	30	98%

Table 6: Success rate of individual *Visual* interactions

Visual Interaction	Successful interaction	Failing interaction	Success Rate
Check	1572	27	98.3%
Click	1240	0	100%
FullCheck	906	2	99.8%
Scroll	869	1	99.9%
TypeText	120	0	100%
Total	4707	30	99.3%

driver on the total amount of interactions in the test suites. We evaluated the percentage of correct interactions also for each type of *Visual* interactions available. *Visual* interactions were classified in five different types of operations: clicks instructions, type instructions, scroll instructions, checks (i.e., verification of the presence of the *Visual* oracle of an individual widget), and full checks (i.e., verification of the *Visual* oracle of a full-screen capture of the AUT).

5.3 Results

All the test suites that were developed involved a certain amount of command involving scroll operations or the *anyOf - AllOf* assertions and were therefore not translatable – by design – from the state-of-the-art version of the TOGGLE tool. In table 4 we compare the absolute amount and the percentage of test cases that were translated for both versions of the tool.

The results highlight that the employed methodology is able to ensure a significant increase in the number of test cases that are translatable to *Visual* test scripts. The usage of scrolling operations allows testing apps that largely rely on dynamic view containers exhaustively.

Figure 5 reports the results obtained to answer RQ2. On the 300 test executions for each test suite, we observed a minimum success rate of 94% for PDF Converter (18 test failures) and a maximum success rate of 100% for Simple Calendar. These results prove that it is possible to obtain *Visual* test suites that have a robustness comparable to *Layout-based* ones.

Figure 6 reports the results obtained to answer RQ3. In the table, the total amount of executed interactions of each type is shown. The number of executed interactions depends on the number of repetitions of the test scripts and the failures during the test run. All interactions that would happen in a successful test run are not executed after a failure. Of the 30 failed instructions, it can be seen that 27 of them were *check* instructions performed on individual locators of the GUIs, 2 were *full checks* (generally performed at the end of a test sequence). Only a single *scroll* interaction was not performed properly and failed because of the inability to correctly reaching the element in the AdapterView, thus making it impossible to proceed with the test case. No *click* or *text input* operation failed. This result was indeed expectable because the *Visual* script runner does not perform any verification when launching mouse and keyboard inputs against the emulated system. If these atomic operations are performed incorrectly, the test case's failure is typically triggered by subsequent check instructions.

6 THREATS TO VALIDITY

6.1 Threats to Construction Validity

We empirically validated the feasibility of translation of gestures on Android components in terms of the success rate of the translation and robustness of the test case executions. An aspect that can be evaluated is the possibility of false negatives, i.e., elements partially made visible by scroll operations that would make *Layout-based* test work but would lead *Visual* scripts to fail. We did not take into account this aspect in the current evaluation of the tool.

6.2 Threats to External Validity

A limitation to the external validity of the evaluation is introduced by the construction of the test suites, in which we used only Espresso methods supported by the translator. Therefore the results of the evaluation are not generalizable to any Espresso test suites. The technique is also not applicable to other testing tools or syntaxes for the Android domain.

As well, *Visual* testing tools for Android apps are typically not applicable to SUTs featuring very dynamic GUIs (e.g., videogames or apps with multimedia content). The evaluation results defined in this paper are therefore not generalizable to SUTs of different categories than those used as experimental subjects. All the applications mined belonged to the *Work and Productivity* category of the F-Droid repository of Android apps.

7 CONCLUSION AND FUTURE WORK

This paper presented a re-engineering of the prototype framework we showed in our previous works to enable automated translation between 2nd generation and 3rd generation GUI test cases for mobile applications.

We added the support of more complex Android-specific gestures, i.e. the scroll interactions, that are very commonly executed when interacting with a mobile app.

To validate the methodology that we defined for the recreation of context-based gestures, we performed the translation on 150 test cases developed for five different experimental subjects. While the

original prototype tool was able to translate correctly only 48 of the test cases (32%), the improved version was able to translate the entirety of the test suite. The generated visual instructions also proved to have very low flakiness and to be repeatedly executable with a very low probability of triggering false negatives when executed by the *Visual* test engine.

This experimental evidence creates the basis for future improvements of the current methodology and tooling. First, we plan to extend the tool to support Kotlin, which is becoming the first-choice programming language to develop Android apps. This extension would require creating a new Enhancer module able to parse Espresso test classes developed in Kotlin. Another planned improvement is the development of additional Enhancer modules to add translation capabilities for other *Layout-based* testing tools, e.g. Appium or UIAutomator. Finally, we plan to evaluate the benefits introduced by the translation-based approach in industrial contexts and empirically evaluate the improvements that can be obtained regarding the maintenance effort of test scripts and the portability of tests to different devices.

ACKNOWLEDGMENTS

The authors would like to thank MSc Vittorio Di Leo for his contribution to this work.

REFERENCES

[1] Emil Alégroth, Zebao Gao, Rafael Oliveira, and Atif Memon. 2015. Conceptualization and evaluation of component-based testing unified with visual gui testing: an empirical study. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 1–10.

[2] 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 *International Symposium on Search Based Software Engineering*. Springer, 3–45.

[3] Luca Ardito, Riccardo Coppola, Maurizio Morisio, and Marco Torchiano. 2019. Espresso vs. eyeautomate: An experiment for the comparison of two generations of android gui testing. In *Proceedings of the Evaluation and Assessment on Software Engineering*. 13–22.

[4] Luca Ardito, Riccardo Coppola, Marco Torchiano, and Emil Alégroth. 2018. Towards automated translation between generations of gui-based tests for mobile devices. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*. 46–53.

[5] Riccardo Coppola, Luca Ardito, Marco Torchiano, and Emil Alégroth. 2021. Translation from layout-based to visual android test scripts: An empirical evaluation. *Journal of Systems and Software* 171 (2021), 110845.

[6] Riccardo Coppola, Maurizio Morisio, and Marco Torchiano. 2018. Mobile gui testing fragility: A study on open-source android applications. *IEEE Transactions on Reliability* 68, 1 (2018), 67–90.

[7] Muhammad Kamran, Junaid Rashid, and Muhammad Wasif Nisar. 2016. Android fragmentation classification, causes, problems and solutions. *International Journal of Computer Science and Information Security* 14, 9 (2016), 992.

[8] Pavneet Singh Kochhar, Ferdian Thung, Nachiappan Nagappan, Thomas Zimmermann, and David Lo. 2015. Understanding the test automation culture of app developers. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 1–10.

[9] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. Droidbot: a lightweight ui-guided test input generator for android. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 23–26.

[10] Mario Linares-Vásquez, Kevin Moran, and Denys Poshyvanyk. 2017. Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 399–410.

[11] Leotta Maurizio, Stocco Andrea, Ricca Filippo, and Paolo Tonella. 2018. Pesto: Automated migration of DOM-based Web tests towards the visual approach. (2018).

[12] Shiwangi Singh, Rucha Gadgil, and Ayushi Chudgor. 2014. Automated testing of mobile applications using scripting technique: A study on appium. *International Journal of Current Engineering and Technology (IJCET)* 4, 5 (2014), 3627–3630.

[13] Jin-lei Sun, Shi-wen Zhang, Song Huang, and Zhan-wei Hui. 2018. Design and application of a Sikuli based capture-replay tool. In *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 42–44.

[14] Denys Zelenchuk. 2019. Espresso and ui automator: the perfect tandem. In *Android Espresso Revealed*. Springer, 165–189.