Translation from Visual to Layout-based Android Test Cases: a Proof of Concept

Translation from Visual to Layout-based Android Test Cases: a Proof of Concept

Riccardo Coppola, Luca Ardito, Marco Torchiano
Politecnico di Torino
Turin, Italy
first.last@polito.it

Emil Alégroth
Blekinge Institute of Technology
Karlskrona, Sweden
emil.alegroth@bth.se

***Abstract*—*Context*: 2nd generation (Layout-based) and 3rd generation (Visual) GUI testing are two approaches for testing mobile GUIs, both with individual benefits and drawbacks. Previous research has presented approaches to translate 2nd generation scripts to 3rd generation scripts but not the vice versa.**

***Goal*: The objective of this work is to provide Proof of Concept of the effectiveness of automatic translation between existing 3rd generation test scripts to 2nd generation test scripts.**

***Method*: A tool architecture is presented and implemented in a tool capable of translating most 3rd generation interactions with the GUI of an Android app into 2nd generation instructions and oracles for the Espresso testing tool.**

***Results*: We validate our approach on two test suites of our own creation, consisting of 30 test cases each. The measured success rate of the translation is 96.7% (58 working test cases out of 60 applications of the translator).**

***Conclusion*: The study provides support for the feasibility of a translation-based approach from 3rd to 2nd generation test cases. However, additional work is needed to make the approach applicable in real-world scenarios or larger open-source test suites.**

***Index Terms*—Testing, Android, Mobile Applications, Software Engineering**

I. Introduction

The user experience provided by Android apps, through their GUI (i.e., Graphical User Interface), now-days rival the experience and complexity previously exclusive to desktop applications. An abundance of features, along with the high competition among apps in the market (e.g., nearly three million apps are available on the official PlayStore as of September 2019[1]), drive a need for more thorough Verification and Validation of Android apps. Additionally, as for traditional software, Android apps are subject to the constant push for faster delivery of software, a trend that is fueled by the widespread adoption of agile approaches to software development. Thus, making automated testing a necessity for the application of practices like continuous integration and delivery.

While low-level, white-box test automation (e.g., unit testing with JUnit) is common practice among mobile developers, higher-level practices like GUI-based testing have a significantly lower diffusion. Often, GUI test cases are not automated, instead they are performed manually with high error-proneness and cost.

Several techniques have been proposed by the literature to execute GUI-based automated testing. It is possible to categorize these techniques under three chronological *generations*. *1st Generation* testing tools (also called Coordinate-based) drive the interaction with the GUI by leveraging exact coordinates of the elements of the screen involved in the test cases. *2nd generation* testing tools (also called Property-based or Layout-based) identify the GUI elements to interact/verify utilizing unique attributes specified in the application layouts, according to the specific programming patterns of the application domain. *3rd generation* testing tools (also called Visual, or Image-recognition based) identify the GUI elements to interact/verify using computer vision and image recognition algorithms that use screen captures of the expected appearance of the application components as input.

1st generation tools have mostly been abandoned since coordinates of GUI elements often change, making them very fragile as locators. 2nd and 3rd generation testing tools, instead, dominate industrial practice and research into them has shown that they have complementary characteristics, and that a combined usage of both the techniques would be beneficial.

However, there are few documented cases in the literature about combined usage of both the techniques in real scenarios.

In our previous work [1], we proposed an architecture for a tool able to translate 2nd generation to 3rd generation test cases, and vice-versa. The rationale behind the research is to identify ways of taking advantage of the commonalities between the two typologies of scripts, e.g., step-wise test sequences and GUI-widget information used at each interaction. The translation of one generation to the other would thereby allow their individual benefits, to counter their respective drawbacks. Such a translation-based approach would also be able to mitigate of one of the most crucial deterrents to the adoption of automated GUI testing, i.e., the high fragility of test suites and the resulting high cost for test case maintenance.

The work presented in this paper builds on the conceptual architecture presented in previous work. Specifically, by detailing and evaluating the implementation of one of its two main features, i.e., the translator from 3rd generation to 2nd generation test scripts. In summary, the contributions of this work are the following:

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

- Proof of Concept, supported by quantitative results from two cases, that 3rd generation test cases can be translated into 2nd generation test cases.
- A description of the approach, and the technical challenges in implementing it;
- A discussion of the possible applications of the approach;
- Ideas for future research within the area of the topic.

The remainder of the manuscript is organized as follows: Section II provides background information about available testing techniques and tools for Android applications, and the reported drawbacks in literature; Section III motivates the current work, describing the benefits provided by a translation-based creation of 2nd generation test cases from 3rd generation test cases; Section IV describes the architecture of the developed tool and provides technical details; Section V reports the results of a first evaluation of the tool on two open-source applications; Section VI discusses related work and similar approaches in the literature; Section VII discusses the threats to the validity of this study; finally, Section VIII concludes the paper.

II. Background

This section describes the typical characteristics of Visual and Layout-based testing techniques when applied to Android applications. This section also overviews the principal tools proposed by the market and the literature.

A. Android Native Apps Structure

It is possible to divide Android apps into three main typologies, based on the way they are developed:

- *Native*: applications developed for Android specifically using the OS constructs and design guidelines;
- *Web-based*: normal web applications optimized for a mobile browser;
- *Hybrid*: applications built using multi-platform web technologies embedded into a native, platform-specific wrapper (e.g., Xamarin or Cordova).

While 3rd generation testing techniques are platform-agnostic and can be applied to any application provided with a pictorial GUI, 2nd generation practices test Android apps according to the definition of the layouts of the app and therefore require native code (white-box techniques) or at least knowledge about Android layout and screen hierarchies (grey-box techniques). It is possible to test Web-based and hybrid apps by using testing tools for web applications.

We focus this work on testing native mobile apps; thereby we will report the typical structure of a native Android application.

In native Android apps, the appearance and behavior for each GUI screen is defined by components named *Activities*. When instantiated, the activities populate the layout of the screen. Layouts can be defined statically in *layout files*, i.e., XML files containing the definition and the properties of the GUI components. All layout files are stored in a fixed location of the Android project (i.e., under the *layout* or *menu* folders). Multiple layouts can be given for the same activity, to provide different arrangements (or appearance) of the widgets on different devices. During the execution of a native app, some particular layouts can be generated dynamically, e.g., when lists of generated elements are to be shown in the GUI, or when widgets are programmatically added to the GUI. 2nd generation testing tool base the interaction with the widgets and the verification of the current state of the screen on the properties that are defined in the layouts, e.g., individual ids, text, parent/child relationships (between widgets and their containers) or content description.

B. Tools for Android testing

A systematic mapping study by Linares-Vasquez et al. [2] has identified many 2nd generation testing tools tailored to work with Android applications. The mapping provides a categorization of the tools based on the way the sequences of interactions are obtained and described. *Automation Frameworks and APIs* provide interfaces to perform operations on widgets of the GUI, and to verify their properties, by manually writing test-scripts in JUnit-like syntax. The two principal testing tools provided by Android, Espresso [3] and UIAutomator [4], belong to this category of 2nd generation testing tools; other tools that achieved relevant diffusion are Appium [5], Robolectric [6], and Robotium [7].

Record & Replay testing tools acquire sequences of interactions that are executed manually on the app's GUI and create repeatable sequences of interactions based on them. These sequences are typically based on the layout properties of the apps to be replayed. Examples of Record & Replay testing tools are Barista [8], RERAN [9] and the official Espresso Test Recorder [10].

The most recent research has focused on automated generations of input sequences. Such sequences can be created through random explorations of the GUIs, as it is done by SAPIENZ [11], CrashScope [12], and Stoat [13], or through model-based representation of the layouts, as it is done by MobiGUITAR [14].

Fewer works in related literature introduced 3rd generation testing tool specific to mobile applications. One exception is provided by SPAG-C, an older proposal which relied on an external physical camera to record the screen captures of the interacted widgets on a physical device. The platform-agnostic nature of 3rd generation testing tools allows, however, the application of any image-recognition based testing engine to Android apps on an emulated device. Compared to 2nd generation ones, these tools provide simpler means of interaction with the GUI, since they only allow to emulate mouse and keyboard operations. In fact, they do not provide special commands specific to Android apps (e.g., means to press the back, home or menu button on the device) that are instead offered by 2nd generation testing tools. Examples of general-purpose 3rd generation tools are SikuliX [15], EyeAutomate [16] (formerly known as JAutomate [17]) and AppliTools [18].

C. Limitations of Android testing

Several studies in the literature have highlighted the issues of existing testing tools for Android applications, and the limited adoption by developers from both the industry and the open-source community. Linares-Vasquez et al. highlighted many common issues for Android testers [19]. Kochhar et al. found – out of a sample of open-source developers – that the diffusion of test automation was scarce, and that developers mostly relied on manual testing or traditional unit testing (with tools like JUnit) instead of applying higher-level testing techniques [20].

Testing fragility is considered one of the main motivation for the limited adoption of testing techniques for mobile tools. A test case is defined *fragile* when it fails during the evolution of the application, because of changes in the locators used by the tests. 2nd generation tests are hence fragile to changes in the layout definition and in the properties of the widgets, while 3rd generation tests are fragile to changes in the pictorial appearance of the widgets. Fragility manifests as an added maintenance costs for test cases. In many cases, such cost represents an important percentage of the total maintenance cost of the whole software project [21].

III. Motivation

Despite the limited penetration of both 2nd and 3rd generation testing techniques among Android developers, there is evidence in the literature about the benefits provided by their adoption. As well, research has highlighted that the combined adoption of both techniques can yield beneficial results, since they have mutual benefits and drawbacks [22].

In our previous conceptualization [1], we have proposed an automated approach based on translation, from one generation of test scripts to the other, and vice-versa. The implementation of the approach thereby requires the development and validation of a module to generate visual oracles/locators from layout properties, and another module to perform the backward translation. The current manuscript aims to provide a proof of concept of the implementation of the latter.

Such 3rd to 2nd translation would provide the following benefits:

- Automate the creation of 2nd generation tests from systematic reuse of existing 3rd generation tests. This automated generation would reduce the development costs and the time for the retrieval of proper layout-based locators and oracles, reducing the need for a manual analysis of the layout files. Such manual exploration of layout files is placed by practitioners among the most time-consuming operations to perform in the creation of Layout-based test cases [23].
- Enhance the expressive power of existing test suites developed with 3rd generation testing tool – that can verify only the pictorial appearance of the app – allowing to verify layout properties of the app. The creation of Java-based test scripts would also allow adding code-related assertions to the translated 2nd generation test cases.

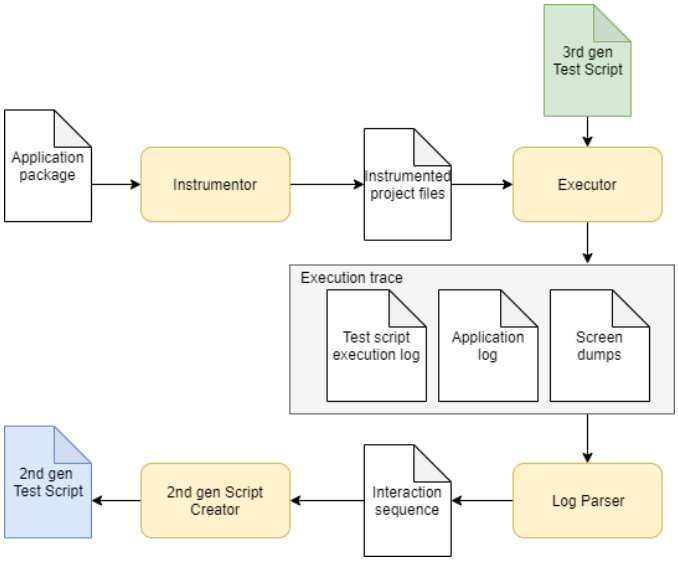

Fig. 1: Architecture of the proposed tool

- Mitigate the fragility of 2nd generation test cases when they are used with regression testing purpose. While 2nd generation test cases are fragile to changes in the definition of the app layouts between different releases, 3rd generation tests remain valid if the pictorial appearance is unchanged. A valid 3rd generation test can hence be used to re-create a working 2nd generation test script when layout-based fragilities are present during the app evolution.

IV. Tool Architecture

The proposed tool architecture is shown in Figure 1. The tool receives as input a (set of) 3rd generation test case(s), and the package of the Android app to test.

The output of the tool is a translated 2nd generation test case. Four distinct modules compose the architecture:

- **Instrumentor**: it is in charge of instrumenting the application code, by adding callbacks to the logging methods and ensuring that each widget is provided with a candidate locator for 2nd generation tests;
- **Executor**: it executes the 3rd generation tests on the instrumented AUT so that the interaction logs can be collected;
- **Log Parser**: it parses the logs obtained when executing the tests, to reconstruct – in a tool-agnostic way – the sequence of performed interactions;
- **2nd generation test creator**: it translates the tool-agnostic sequence of interaction to the selected 2nd generation syntax.

This conceptual architecture can be tailored to work with different testing tools and syntaxes. For our purposes, we selected EyeAutomate as the starting 3rd generation testing tool, due to our previous experience with the tool, and Espresso as the destination 2nd generation testing tool, due to its high diffusion in open-source Android projects [24].

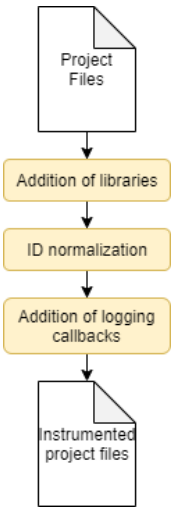

Fig. 2: Steps performed by the Instrumentor module

The following subsections provide details about the implementation choices we took to realize each of the four modules.

A. Instrumentor

The Instrumentor has the role of initializing the project. It obtains as input the folder with the project files that have to be instrumented and gives as output a folder of temporary project files to be used to generate 2nd generation test cases. In this module, we used the JavaParser[2] open-source tool to parse and modify the Java files of the application.

Figure 2 shows the flowchart with the steps performed by the Instrumentor. The description and motivation of the steps is given in the following.

1) Addition of libraries: At first, the module verifies the presence of the needed support Android libraries, such as the RecyclerView and SupportPreference. Also, it analyzes the application package Gradle file to uniform the imports of all applications to the new version of the support library, AndroidX. Hence, the module maps all references to the support library packages into the corresponding androidx.* packages.

2) ID Normalization: In this step, the module analyzes all the layout files in the *res* or *menu* folder, to find widgets without unique IDs. For those widgets, the module adds a new programmatically-generated ID. The addition of unique IDs is required because, in some cases, it is not possible to retrieve a unique property to identify a widget in 2nd generation test cases [23]. This step guarantees that, by construction, all the widgets in the app have unambiguous locators or oracles for a 2nd generation test case.

3) Addition of logging callbacks: In this step, the module adds the required callbacks to the production code of the AUT. The application logger generates logs for each interaction with the application caused by the execution of the EyeAutomate test script. This action will help in the reconstruction of the interaction sequence. The module defines a custom class, called *MyWindowCallback*, which extends *android.view.Window.Callback*, to intercept click or swipe operations performed on the GUI of the AUT. The management of type instructions will be based on information gathered from the 3rd generation execution log. The instrumentor adds the definition of the MyWindowCallback inside the *onStart* method of each activity in the application package.

In the custom Callback, the tool modifies the *onTouchEvent* method to intercept the operations *ACTION_DOWN* (i.e., when the user touches the screen), and *ACTION_UP* (i.e., when the user releases the screen).

Specifically, every time the module intercepts an ACTION_DOWN event, it performs the following operations:

1) checks whether the operation is the first in the tap or swipe sequence (long clicks or swipes will cause other executions of the onTouchEvent callback);
2) stores the event time;
3) stores the event coordinates;
4) calculates the interacted view, as the uppermost clickable view in the hierarchy whose bounding rectangle contains the coordinates of the interaction.

Every time the module intercepts an ACTION_UP event, it checks whether the event is a tap or a swipe by checking the distance between the starting and ending position and whether it is a short or long-tap by comparing the starting and ending time of the interaction.

The module also edits three other callbacks in the MyWindowCallback class to cope with special cases discussed in the next subsection. The *onWindowFocusChanged* callback is used to check the current fragments on screen; the *dispatchedKeyEvent* is used to intercept the back button press; the *onDetachedFromWindow* is used to intercept dialog dismissal.

4) Special Cases: Several special widgets required additional effort since they were not manageable by the usage of the MyWindowCallback class. We list them in the following:

- *Dialogs*: A Dialog is a window that prompts information to the user by floating on the rest of the GUI content (e.g., Alerts, Time and Date pickers). The Dialog can be created by a specific Builder class (like in the case of objects of the MaterialDialog and AlertDialog classes) or without a specific Builder class (like the DatePicker and TimePicker dialog). To intercept clicks on the buttons to close Android dialogs, the *onDetachedFromWindow* method of the dialog class must be modified, because it is called before the *onTouch* method of the window callback. The Instrumentor hence searches for classes extending known dialog types to add logging instructions on the *onDetachedFromWindow* callback.
- *Preferences*: Preferences allow the user to change functionality and behavior of an application and are typically shown in the Settings menu. To be managed, Preferences required an extension of the *onPreferenceChanged* callback.

[2]https://github.com/javaparser/javaparser

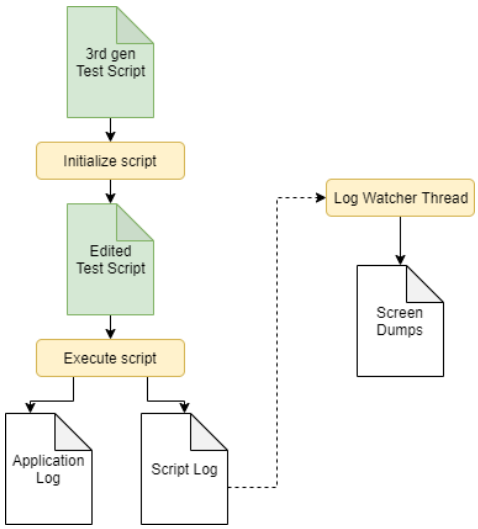

Fig. 3: Steps performed by the Executor module

```
2019-07-31 18:26:43  Click "{ImageFolder}/1564247147926.png" 43 45
2019-07-31 18:26:44  Sleep 5000
2019-07-31 18:26:50  Click "{ImageFolder}/1564247169961.png" 85 42
2019-07-31 18:26:52  Sleep 5000
2019-07-31 18:26:58  Click "{ImageFolder}/1564273167953.png" 17 80
2019-07-31 18:27:00  Sleep 5000
```

Fig. 4: Sample log of the 3rd generation test

- *Spinners*: Spinners provide a way to select a value from a set, always showing only the selected one. The method to be edited to manage the selection of a Spinner is *onItemSelected*.
- *Fragments*: Android Fragments are portions of the screen that can be instantiated at runtime inside a started Activity, and are managed by the FragmentManager. The *onAttachFragment* callback is in charge of managing the rendition on the screen of the Fragment. Since, in some cases, such callback is not called inside the *onStart* callback of the container activity, we modified the *onAttach Fragment* itself, to explicitly add the Fragment to the screens managed by the custom MyWindowCallback.
- *Overflow menus*: In some cases, the menus prompted to the user are available in a window separated from the activity. In these cases, the menu items do not have a unique id, and we need to log the text as a locator. Also, the closure of the menu requires the creation of a custom *onPanelClosed* callback to log the information about it.

B. Executor

The Executor module is in charge of preparing and launching the 3rd generation starting test script, executing its instructions on an instrumented AUT emulated on an Android Virtual Device (AVD). The execution of the script on the instrumented AUT allows the generation of two different logs: (1) an Application Log, created thanks to the callbacks injected by the Instrumentor, and (2) a Script Log, created by the 3rd generation test runner. The flowchart in Figure 3 shows the

```
07-31 18:26:43.741 18293 18293 D Test_logger: SHORT_CLICK org.ligi.passa
ndroid:id/fab expand menu button 0.48954815 0.4661559
07-31 18:26:51.405 18293 18293 D Test_logger: SHORT_CLICK org.ligi.passa
ndroid:id/fab_action_create_pass 0.4422866 0.40217057
07-31 18:26:59.691 18293 18293 D Test_logger: SHORT_CLICK android:id/lis
t 3 0.16674805 0.576527
```

Fig. 5: Sample application log

TABLE I: Types of logged actions

Action Type	Description
SHORT_CLICK	Normal click on a widget (press time ¡ 400ms).
LONG_CLICK	Long click on a widget (press time ¿ 400 ms).
SWIPE	Swipe, or drag and drop operations.
BACK	Used when the back button is pressed during the navigation.
SPINNER	To indicate operation on a spinner. In this case, the text is used as a locator.
PREFERENCE	To indicate operation on a preference element. In this case, the text is used as a locator.
MENU	To indicate the opening of an overflow menu. In this case, the text is used as a locator.
MENU_DISMISSED	Called when the overflow menu is closed.
DIALOG_CANCEL	Called when a dialog is canceled.

artifacts and the operations performed by the Executor. Details about each step and element are given in the following.

1) Initialization: Before the execution of the 3rd generation test script, the Executor module initializes by creating a temporary version of the script with sleep instructions (of one second) added between each command. The addition of sleep instructions allows us to easily differentiate between each operation (referred to just with its timestamp) and to make time for the generation of the screen dumps of the AUT.

The initialization phase also launches a thread (the *Log Watcher*) in charge of reading the log generated by the 3rd generation testing tool and dumping the screen of the Android applications. The motivation and details of the dumping operations are provided in a later subsection.

2) Script execution and script log: The Executor launches the application on the emulated device and then runs the 3rd generation test case against it.

3rd generation automation tools create logs with the trace of the operations performed and its details. Each log line is composed of a timestamp, and information about the executed operation (type, referred screen capture, and relative coordinates inside the screen capture; see Figure 4 for a sample log).

The Log Watcher thread is executed to monitor changes in the script log, to determinate when the 3rd generation tests perform check operations. Such external monitoring is necessary because 3rd generation checks verify the graphical appearance of the rendered GUI without performing any interaction on the emulated device. It is hence impossible to trigger callbacks to record the screen state in the emulated applications. To obtain information for the assertions, we hence triggered an external command – from another thread, named *LogWatcher* –

to dump the information about the current screen when a check operation is appended in the 3rd generation script log. To obtain the screen dump (i.e., an XML file containing information about all the views that are currently visible) we leveraged the UIAutomator tool.

3) Generation of the application log: The application log is generated by the application package executed on the Android Virtual Device, thanks to the custom callbacks inserted by the Instrumentor module. We used the built-in Android LogCat tool for creating the logs. A LogCat line is composed by three elements:

1) the timestamp, useful to associate each operation from the 3rd generation script to the related application log-line;
2) a tag used to filter out the log of the AVD;
3) the string containing the log message used to store the required information.

Figure 5 reports an excerpt from the application log.

A logline contains the following information:

- *Action:* the type of action performed in the AUT. It is the only mandatory component of a logline. The types of logged interactions are reported in Table I.
- *Locator:* the locator used to identify the widget when the 2nd generation test will be created. The locator can be an ID, a text value, or a list of numeric indexes followed by an ID. IDs are used for widgets that had a unique ID in the original application package, or for widgets that were assigned a unique ID by the Instrumentor.
  The text value is used for objects like Preferences or Spinners, whose elements cannot be assigned a unique ID with the Instrumentor, because of values that can be added programmatically at runtime.
  Finally, lists of indexes are used when the view does not have an ID nor a unique text value: the indexes represent the parent-child connection between the view and the first parent with a unique id. The unique id of the parent is then attached at the end of the string. This representation is useful in the case of RecyclerViews and AdapterViews populated at runtime.
- *Coordinates:* the position of the click inside the view, represented as the offset in percent from the center on both x and y axes. Logging the coordinates inside the widget allows to perform clicks in different points of the button than the center.
- *Distances:* used in case of swipes or drags, the distances on both x and y axes from the starting position.

C. Log Parser

The Log Parser module is in charge of generating a tool-agnostic list of objects, that can later be used to generate a 2nd generation test script. The module uses as input the script log, the application log, and the related screen dumps. It also receives the screen size of the emulator that launches the app. The flowchart in Figure 6 shows the artifact and the operations performed by the module. We provide details in the following.

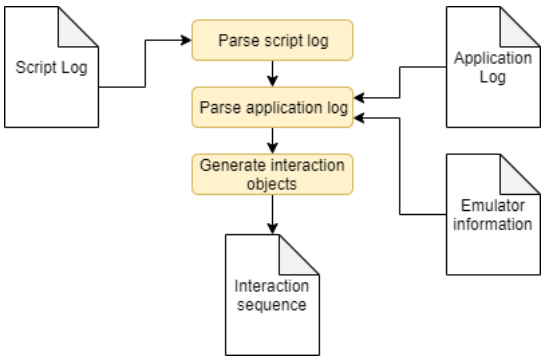

Fig. 6: Steps performed by the Log Parser module

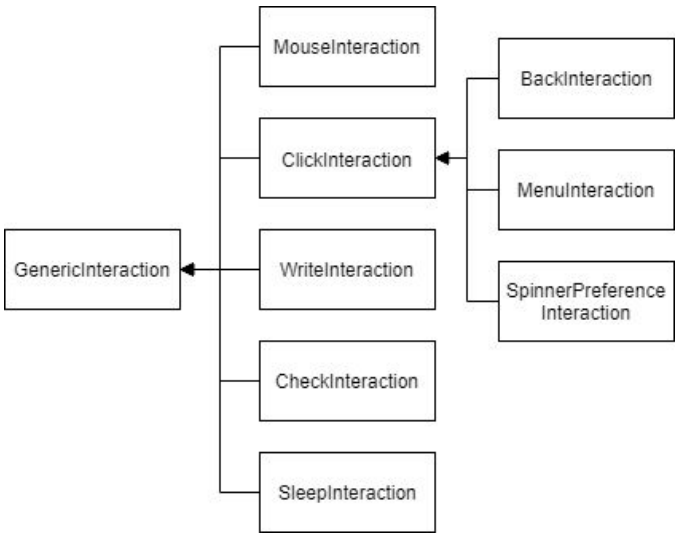

Fig. 7: Types of interactions created by the Log Parser

1) Script log parsing: Firstly, the Log Parser parses the script log, generated by the 3rd generation test driver, and deduces the list of the performed operations. The parser searches in the script log for the commands that could be used in the testing of an Android application: Click, DragStart, DragDrop, Check, MouseLeftPress, MouseLeftRelease, Type, Sleep.

For each parsed command, the module creates an InteractionObject. Different subclasses are defined to represent the various types of interactions that are considered by the translator since different data is required based on the type of command.

2) Application log parsing: Secondly, the Log Parser parses the application log, generated by the custom MyWindowCallback when it intercepts clicks on the emulated AUT. The logged timestamp is used to find correspondence between lines in the script log and the application log. In the case of check operations, the module needs to retrieve the locator information about the checked view from the dump file with the related timestamp.

3) Interaction Objects: Interaction Objects created and populated by the Log Parser are summarized in the diagram

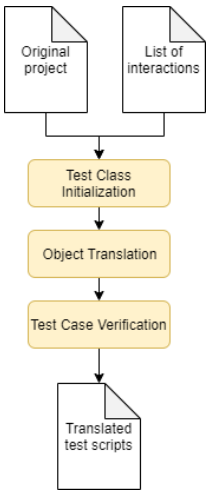

Fig. 8: Steps performed by the 2nd generation Script Creator module

in Figure 7 and detailed in the following:

- *ClickInteraction*: it represent click operations, and can be categorized in different subclasses:
  - *Back*: represents a click on the Back button, i.e., back navigation in the app;
  - *Spinner-Preference*: represents a click that selected a spinner or preference value, that is specified in the object;
  - *Menu*: represents a click that opened an overflow menu item, or an item inside the menu. The locator of the click can be either a string (the value in the menu) or null (if the click represents the menu opening);
  - *Normal*: represents a short or long click on a view.
- *MouseInteraction*: it is used to represent swipe operations. The swipe is not limited to be horizontal or vertical but can have custom angles.
- *WriteInteraction*: it represents a type operation on a TextView. The object contains a string attribute with the text to be typed in the view.
- *CheckInteraction*: it represents a check operations in the 3rd generation script, that has to be translated into an assertion in the 2nd generation test.
- *SleepInteraction*: it represents a sleep operation, that has to be translated into an equivalent sleep instruction in the code of the created 2nd generation test.

D. 2nd generation Script Creator

The 2nd generation Script Creator is the last module of the architecture. It receives a list of objects that must be translated into 2nd generation method calls. The steps performed by the module are shown in Figure 8 and detailed in the following.

1) Test Class Preparation: The first operation of the Script Creator module is the creation of the test classes where the test scripts will be translated. The tool maps 3rd generation interaction sequences to 2nd generation test methods, inside a single test class. The creation of the test class is performed with the JavaParser tool.

The test class is placed in the *androidTest* folder of the Android project (i.e., the default one used by GUI testing tools like Espresso or UI Automator). Each test class is enriched by a series of custom methods that are needed for the execution of the translated operations:

- *childAtPosition*: the method is used to match a view in the hierarchy inside a dynamically populated list of views, identified by its index among all the children view of a given parent;
- *childAtPositionCheck*: similar to the previous method but used for check operations, in case the list of views had been scrolled from the original position;
- *typeMatcher*: used to match a view to input text. It is required because in the translated script click, and type operations are separated into different method calls;
- *coordinatesFunction*: the method returns the coordinates on which a view has to be clicked, in case the click does not have to be performed in its center.

2) Object Translation: After completing the initialization procedure, the module translates the individual operations to calls to Espresso methods.

We report the translation of the supported interactions in Table II. For readability reasons, we do not provide for each method the full list of parameters.

Interaction Type	Espresso method
MouseInteraction	GeneralSwipeAction(Swipe.FAST)
Click	GeneralClickAction(Tap.SHORT)
Long Click	GeneralClickAction(Tap.LONG)
Menu Open	openContextualOverflowMenu
Menu click	onView(withText(...)).perform(click())
Back	ViewActions.pressBack())
Spinner/Preference	onView(withText(...)).perform(click())
Check	check(matches(isDisplayed()))
Sleep	Thread.sleep(3000)

TABLE II: Caption

Once the module completes the translation, it saves the test, and it builds the application package. The module also checks the outcome of the failure in order to avoid saving non-working test scripts.

V. Tool Evaluation

This section describes the experimental evaluation that we performed to verify the feasibility of the approach implemented by the tool.

A. Experimental Subjects and Setup

We chose two open-source applications to evaluate the tool performance: OmniNotes [3] (v6.0.0 Beta 7), a note management application, and PassAndroid [4] (2.5.0), an electronic tickets manager. The code of both applications is on GitHub, and both of them are available on PlayStore as well. We

[3]http://github.com/federicoiosue/Omni-Notes
[4]http://github.com/ligi/PassAndroid

TABLE III: Characteristics of selected apps (as of September 2019)

	OmniNotes	PassAndroid
Number Of Downloads	100,000+	1,000,000+
Number Of Releases	120	100
Tested release	6.0.0 Beta 7	2.5.0
Java LOCs	48,116	32,309
Number of Activities	13	17
Number of Layout Files	52	19

chose these two applications for their different appearances and behavior. Their characteristics can be found in Table III.

For the evaluation, we wrote two test suites of 30 EyeAutomate test cases each in order to cover most of the apps' features and widgets. We designed each test to be executable independently for avoiding cascading failures. All test cases start from the Main application activity. The scripts contain from 2 to 43 commands, mostly short clicks, and checks.

We run the test cases on a laptop PC with an Intel i7-6700HQ CPU at 2.60GHz clock, with 16GB RAM and Window 10 Operating System. We performed the test suites development and the EyeAutomate test cases execution in EyeStudio 2.1. We launched the apps on an emulated Nexus 6P API 26 (Android 8.0) with a disabled device frame and disabled animations.

B. Procedure

The experimental evaluation aimed to answer the question:

RQ What is the success rate of the layout-based test scripts generated through translation?

To answer the question for each of the two applications, we computed the Success Rate (SR), defined as:

$$SR_t = \frac{N_s}{N_{ex}}, \tag{1}$$

where N_s is the number of generated test scripts whose execution ended with a success, and N_{ex} is the total number of scripts that we attempted to translate with the tool. Failed tests can be of two types:

- The tool failed to generate the test;
- Test was generated but failed its execution.

C. Experimental Results

The experiment had a quite high success rate (96.7%, 29/30, in both test suites with a total of 96.7%, 58/60).

Two failures occurred with the test suites selected for the evaluation. One of the translations for the OmniNotes suite failed during the generation step. The cause was due to the keyboard closure without any input. The tool did not recognize that situation because of a missing implementation of the keyboard closing operation when no input is provided into a TextView. This behavior represents a systematic issue that requires engineering effort to be fixed in future versions of the tool.

A correctly generated test case, which failed during the execution, caused the failure in the PassAndroid test suite. The script tried in fact to type inside a focused view, but no view was focused during the script execution. The fix of this type of error may require more careful handling of focus on new activities during the execution of the obtained Espresso test cases.

VI. Related Work

Several Record & Replay tools, like the Espresso Test Recorder [10] embedded in the Android Studio IDE, or Barista [8], are available for testing Android apps. These testing tools typically create sequences of layout-based interactions. The added value of the proposed approach is to reuse existing working visual test suites, generating 2nd generation assertions automatically based on the visual checks already present in the 3rd generation scripts. Normal 2nd generation testing tools, on the other hand, usually require the tester to manually select assertions to add after interactions with the GUI are performed.

Several approaches have been proposed in the literature to repair broken test cases upon evolution of the AUT [25]. Most of them base the repair of the broken locators on a model-based representation of the app GUI [26] [27]. Our approach would base, instead, the repair of broken 2nd generation locators on another existing representation of the testing tools, with no need of a model (either manually or automatically obtained) of its GUI. Obviously, the repair-based approach remains applicable also when no visual test equivalents are available.

Leotta et al. proposed a translation-based approach for the migration of 2nd generation, DOM-based test cases to visual tests with Sikuli [28]. The approach is similar to the 2nd to 3rd generation translator that we conceptualized for mobile apps in our previous works [1]. The main difference between the approach by Leotta et al. and ours is that the former is only working in one direction, without any automated translation from 3nd to 2rd generation test cases. To the best of our knowledge, no equivalent of our approach exists in the literature.

VII. Threats to Validity

A. Threats to Construction Validity

We measured the tool effectiveness in terms of success rate. A test translation is successful if the tool can complete it and if it is possible to execute the generated test entirely.

Another aspect would be making sure that the translated Espresso script executes the correct instructions in order to avoid false positives. False positives can be the result of wrong translations of 3rd generation interactions to 2nd generation method calls, that however do not invalidate any of the assertions in the test cases.

We did not take into account this aspect in the current evaluation.

B. Threats to External Validity

The experimental design includes some bias as we only used interactions and dialogs supported by the translator. Therefore the results of this evaluation are not generalizable to any EyeAutomate test suite.

Apps with a very different graphical appearance or logic or used dialogs types may induce results that vary significantly from those reported.

Also, 2nd generation tests may not be obtained for applications using complex custom components or re-definitions of dialogs/fragments that would not be intercepted during the project instrumentation.

It is worth mentioning also that the addition of sleep instructions may render the usage of the proposed technique not feasible for timing-sensitive test cases.

VIII. Conclusion and Future Work

The results of the evaluation that we performed show that the proposed tool can provide benefits in terms of an effective translation from 3rd generation to 2nd generation test scripts. As explained in the previous sections, however, several issues are still open from the point of view of the tool implementation. For instance, by now the tool provides a limited coverage of Preferences, Menus and Spinners, on which the interactions can be recorded only when different textual content is provided for each element of the list.

The tool also will need extensive evaluation on a bigger set of software objects, possibly with already existing test suites to avoid biases and increase the external validity of the study. We also plan to add empirical studies on the capability of the proposed proof of concept to reduce the fragility issue of layout-based test suites, and hence to quantify the reduction of maintenance costs for testers/developers.

As final steps of our research plan, we foresee the full implementation of our translation-based architecture in both directions. Such tool, as conceptualized in our first work [1], would enable gaining the benefits of both generations whilst mitigating the costs and drawbacks of the individual approaches. This complete tool would hence reduce the maintenance cost of both generations of testing techniques, while enriching the bug-finding power providing verification of both the layout properties and the graphical appearance of the app widgets.

The present paper also provides a preliminary implementation of the technique, that is only applicable to two specific tools (namely, EyeAutomate and Espresso). Future extensions of this research will involve other testing tools allowing comparisons between them, and enhanced applicability of the technique.

References

[1] L. Ardito, R. Coppola, M. Torchiano, and E. Alégroth, "Towards automated translation between generations of gui-based tests for mobile devices," in *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*. ACM, 2018, pp. 46–53.

[2] M. Linares-Vásquez, K. Moran, and D. Poshyvanyk, "Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing," in *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE, 2017, pp. 399–410.

[3] G. Nolan, "Espresso," in *Agile Android*. Springer, 2015, pp. 59–68.

[4] D. Zelenchuk, "Espresso and ui automator: the perfect tandem," in *Android Espresso Revealed*. Springer, 2019, pp. 165–189.

[5] G. Shah, P. Shah, and R. Muchhala, "Software testing automation using appium," *International Journal of Current Engineering and Technology*, vol. 4, no. 5, pp. 3528–3531, 2014.

[6] B. Sadeh, K. Ørbekk, M. M. Eide, N. C. Gjerde, T. A. Tønnesland, and S. Gopalakrishnan, "Towards unit testing of user interface code for android mobile applications," in *International Conference on Software Engineering and Computer Systems*. Springer, 2011, pp. 163–175.

[7] H. Zadgaonkar, *Robotium Automated Testing for Android*. Packt Publishing Ltd, 2013.

[8] M. Fazzini, E. N. d. A. Freitas, S. R. Choudhary, and A. Orso, "Barista: A technique for recording, encoding, and running platform independent android tests," in *Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on*. IEEE, 2017, pp. 149–160.

[9] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "Reran: Timing-and touch-sensitive record and replay for android," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 72–81.

[10] S. Negara, N. Esfahani, and R. P. Buse, "Practical android test recording with espresso test recorder," in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*. IEEE Press, 2019, pp. 193–202.

[11] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 94–105.

[12] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk, "Crashscope: A practical tool for automated testing of android applications," in *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on*. IEEE, 2017, pp. 15–18.

[13] 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 *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 245–256.

[14] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "Mobiguitar: Automated model-based testing of mobile apps," *IEEE software*, vol. 32, no. 5, pp. 53–59, 2015.

[15] T. Yeh, T.-H. Chang, and R. C. Miller, "Sikuli: using gui screenshots for search and automation," in *Proceedings of the 22nd annual ACM symposium on User interface software and technology*. ACM, 2009, pp. 183–192.

[16] E. Alégroth, *Visual GUI Testing: Automating High-level Software Testing in Industrial Practice*. Chalmers University of Technology, 2015.

[17] E. Alegroth, M. Nass, and H. H. Olsson, "Jautomate: A tool for system- and acceptance-test automation," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 2013, pp. 439–446.

[18] K. F. Hasselknippe and J. Li, "A novel tool for automatic gui layout testing," in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2017, pp. 695–700.

[19] M. Linares-Vásquez, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk, "How do developers test android applications?" in *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE, 2017, pp. 613–622.

[20] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo, "Understanding the test automation culture of app developers," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2015, pp. 1–10.

[21] R. Coppola, M. Morisio, and M. Torchiano, "Mobile gui testing fragility: A study on open-source android applications," *IEEE Transactions on Reliability*, 2018.

[22] E. Alégroth, Z. Gao, R. Oliveira, and A. Memon, "Conceptualization and evaluation of component-based testing unified with visual gui testing: an empirical study," in *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*. IEEE, 2015, pp. 1–10.

[23] L. Ardito, R. Coppola, M. Morisio, and M. Torchiano, "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*. ACM, 2019, pp. 13–22.

[24] R. Coppola, M. Morisio, M. Torchiano, and L. Ardito, "Scripted gui testing of android open-source apps: evolution of test code and fragility causes," *Empirical Software Engineering*, pp. 1–44, 2019.

[25] J. Imtiaz, S. Sherin, M. U. Khan, and M. Z. Iqbal, "A systematic literature review of test breakage prevention and repair techniques," *Information and Software Technology*, vol. 113, pp. 1–19, 2019.

[26] X. Li, N. Chang, Y. Wang, H. Huang, Y. Pei, L. Wang, and X. Li, "Atom: Automatic maintenance of gui test scripts for evolving mobile

applications," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2017, pp. 161–171.
[27] N. Chang, L. Wang, Y. Pei, S. K. Mondal, and X. Li, "Change-based test script maintenance for android apps," in *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2018, pp. 215–225.
[28] M. Leotta, A. Stocco, F. Ricca, and P. Tonella, "Pesto: Automated migration of dom-based web tests towards the visual approach," *Software Testing, Verification And Reliability*, vol. 28, no. 4, p. e1665, 2018.