Translation from Layout-based to Visual Android Test Scripts: an Empirical Evaluation

Riccardo Coppola[a], Luca Ardito[a], Marco Torchiano[a], Emil Alégroth[b]

[a] *Department of Computer and Control Engineering, Polytechnic University of Turin, Italy. e-mail: first.last@polito.it*
[b] *Department of Software Engineering, Blekinge Institute of Technology of Karlskrona, Sweden. e-mail: emil.alegroth@bth.se*

Abstract

Mobile GUI tests can be classified as layout-based – i.e. using GUI properties as locators – or Visual – i.e. using widgets' screen captures as locators –. Visual test scripts require significant maintenance efforts to be kept aligned with the tested application as it evolves or it is ported to different devices.

This work aims to conceptualize a translation-based approach to automatically derive Visual tests from existing layout-based counterparts or repair them when graphical changes occur, and to develop a tool that implements and validates the approach.

We present TOGGLE, a tool that translates Espresso layout-based tests for Android apps to Visual tests that conform to either SikuliX, EyeAutomate, or a combination of the two tools' syntax. An experiment is conducted to measure the precision of the translation approach, which is evaluated on maintenance tasks triggered by graphical changes due to device diversity.

Our results demonstrate the feasibility of a translation-based approach, show that script portability to different devices is improved (from 32% to 93%), and indicate that translation can repair up to 90% of Visual locators in failing tests.

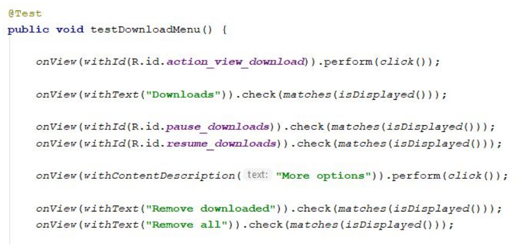
GUI test translation mitigates challenges with Visual tests like maintenance effort and portability, enabling their wider use in industrial practice.

Keywords: GUI Testing, Mobile testing, Empirical Software Engineering, Software Validation.

1. Introduction

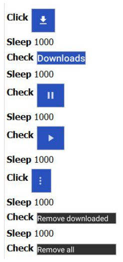
The Android operating system has recently reached its ninth release and has been confirmed as the platform of choice for nearly 90% of mobile users as of the first half of 2019. Modern Android applications (henceforth referred to as apps) are complex, generally on par with desktop software with interactive graphical user interfaces (GUI) and large-scale server back-ends. Similar to desktop software, apps are also developed using modern development processes in quick and short delivery cycles. Short deliveries that make quick, and thorough, verification and validation phases crucial in both open-source and industrial settings. Android apps are also GUI-intensive, putting emphasis on testing their visual correctness in addition to their functional behaviour.

During the last ten years, many end-to-end (from now on referred to as *E2E*) testing tools have been proposed for Android app testing. E2E tests are defined as repeatable test scripts that automate the interaction with the application as a whole, without isolating its components (i.e. a black-box approach), to emulate operations that a human user would perform [1].



```
@Test
public void testDownloadMenu() {

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

    onView(withText("Downloads")).check(matches(isDisplayed()));

    onView(withId(R.id.pause_downloads)).check(matches(isDisplayed()));
    onView(withId(R.id.resume_downloads)).check(matches(isDisplayed()));

    onView(withContentDescription( text: "More options")).perform(click());

    onView(withText("Remove downloaded")).check(matches(isDisplayed()));
    onView(withText("Remove all")).check(matches(isDisplayed()));
```

(a) Sample 2nd generation test script



```
Click
Sleep 1000
Check Downloads
Sleep 1000
Check
Sleep 1000
Check
Sleep 1000
Click
Sleep 1000
Check Remove downloaded
Sleep 1000
Check Remove all
```

(b) Sample 3rd generation test script

Figure 1: Examples of 2nd and 3rd generation test scripts. The test scripts perform the same interaction and checks

These tools fall into one of three generations of testing tools, as defined in the literature. 1st generation tools are the oldest ones, in which the interaction with the user interface is guided by exact screen coordinates as locators of GUI objects. However, scripts developed with these tools have low robustness to GUI change, leading to large maintenance costs, and are therefore seldom used in practice. 2nd generation tools instead use widget properties as locators or oracles for assertions (see fig. 1a).

In the case of Android applications, typical 2nd generation tools use the widget properties specified in XML layout files as locators, e.g., unique identifiers, text content, content descriptions. However, because of the reliance on widget property access, these tools are limited to testing applications written in specific programming languages and are not able to test, for instance, dynamic content (e.g. animations, video or real-time content such as games).

Because of the limitations of 2nd generation tools, 3rd generation testing tools have been proposed that use image recognition technology to test the apps' visual appearance or, more commonly, their behaviour through the pictorial user interface. These script-based test scenarios therefore include screen captures (see fig. 1b) that are used as locators to identify widgets. The screen captures are also used as oracles that compare the current appearance of the app to the visually expected result after the interactions are performed. Because 3rd generation tools rely on image recognition, they are, in contrast to 2nd generation tools, agnostic to platform/system/programming language, requiring only access to the pictorial GUI of the SUT to run tests. However, compared to 2$^{\text{nd}}$ generation, because of the computationally heavy and imprecise image recognition algorithms, tools of this approach generally have lower test execution time performance and lower robustness to graphical change.

2$^{\text{nd}}$ and 3$^{\text{rd}}$ generation testing tools currently coexist in practice of the testing community, although 2$^{\text{nd}}$ generation tools are more common than 3$^{\text{rd}}$ generation ones. We also stress that the adopted categorization of the generations of GUI testing approaches is strictly chronological and do not reflect that later generations should be more effective/efficient. For instance, image-recognition based tools (3$^{\text{rd}}$ generation) are not considered more effective/efficient or a replacement for Layout-based (2$^{\text{nd}}$ generation) tools for GUI-based testing. Research into 2$^{\text{nd}}$ and 3$^{\text{rd}}$ generation tools has instead shown that they have complementary benefits and characteristics [2].

Regardless of generation, automated GUI testing is a costly practice, both in terms of required development and maintenance efforts. These costs prohibit companies from combining generations of techniques and companies instead tend to focus on 2$^{\text{nd}}$ generation tools, complemented with manual testing of the GUI's visual appearance.

The use of 2$^{\text{nd}}$ generation tools for mobile software development can also be explained by the availability of 2$^{\text{nd}}$ generation tools and a gap in research and lack of availability of 3$^{\text{rd}}$ generation tools. In particular, and to the best of our knowledge, no study has explored the complementary benefits of paired use of 2$^{\text{nd}}$ generation and 3$^{\text{rd}}$ generation tools on Android apps similar to desktop applications [2]. However, Android GUI testing seems well-suited for a paired testing approach because of the close coupling between app functionality and GUI appearance. This coupling results in both layout-based and visual locators to frequently change, and therefore require frequent testing [3].

Additionally, Android apps are developed to be used on a myriad of different mobile devices, with varying properties, such as pixel density, resolution or screen ratio [4][5]. This presents a challenge for Visual testing since image recognition algorithms are sensitive to changes in size of expected images. As such, Visual tests developed on one device might not be portable to another, which in practice multiplies the cost of visual test script management by the number of devices on which the tests need to be executed.

Unfortunately, these issues, in particular cost and robustness issues, have proved to be deterrents for the broad adoption of automated GUI testing among Android developers [6]. Thus, presenting a need for research and development into more efficient approaches for the creation of effective (robust) Visual tests.

In this paper, we investigate the creation of 3$^{\text{rd}}$ generation Android app test scripts by translating them from 2$^{\text{nd}}$ generation test scripts that are used as templates. We base this approach on the premise that 2$^{\text{nd}}$ and 3$^{\text{rd}}$ generation tools share several commonalities in terms of the test structure, such as step-wise test sequences of interactions/assertions of widgets, similar timing constraints and similar test purpose for functional tests. The main objective of this work is therefore to improve the value of existing 2$^{\text{nd}}$ generation test scripts by providing practitioners with a cost-efficient extension of their existing 2$^{\text{nd}}$ generation testing capabilities to 3$^{\text{rd}}$ generation testing as well.

This manuscript introduces our approach, which is implemented in a tool called TOGGLE (Translation Of Generations of UI tests at Low Effort). The tool uses 2^{nd} generation Espresso test scripts as templates to create 3^{rd} generation EyeAutomate, SikuliX or mixed scripts. However, the approach is theoretically adaptable and applicable for any pair of testing tools of the two generations.

In summary, this paper provides the following advances to the current state of the art in the field of automated GUI testing for mobile apps:

- A test creation approach built on the translation of 2^{nd} to 3^{rd} generation GUI test cases for Android apps. GUI test translation has previously been demonstrated for Web-based applications [7] but, to the best of our knowledge, not for Android apps except for our previous work that served as the basis of the work presented here [8];
- The general architecture and implementation details of a tool that demonstrates the approach, TOGGLE. The implementation details are complemented with an extended proof of concept study of the tool based on previous research [9];
- Results from evaluation of the success rate of the implemented approach in the translation of 2^{nd} generation test scripts to 3^{rd} generation scripts, the translated tests' execution success rate on Android apps and the ability of the approach to mitigate graphic and fragmentation-induced fragility.
- As a side-effect of this work, the study shows how existing 3^{rd} generation testing tools can be applied to Android applications, which provides a contribution to 3^{rd} generation software testing research [10].

The paper is organized as follows: Section 2 provides background on the different generations of testing approaches, and a review of available testing tools for Android apps; Section 3 provides additional motivating details about the adoption of a translation-based approach; Section 4 describes the architecture and the implementation details of the TOGGLE translator; Section 5 describes the experiments that we conducted to evaluate the feasibility and benefits guaranteed by such an approach; Section 6 discusses the implications of the results and the current limitations of the approach; Section 7 analyzes related work available in the literature; Section 8 concludes the manuscript and lists possible prosecutions of the work.

2. Background

In this section, we describe the basic concepts of the 2^{nd} and 3^{rd} generation testing tools, the available tools for testing Android apps, and the challenges they expose to developers and testers.

2.1. Layout-based (2^{nd} generation) testing of Android apps

Second generation (or Layout-based) testing tools are based on a model of the graphical user interface, that is decomposed in layouts and hierarchies of components. Properties and values are associated with each component of the GUI, allowing the properties to be used as locators (to identify widgets throughout the test cases) or as oracles (to verify the outcome of a test scenario based on widget state). In the case of Android testing, the 2^{nd} generation GUI testing tool leverages the properties defined in the XML layout files to that extent. This type of information describes Android screens as they are organized using the Android application framework peculiarities. However, it does not give insights about the actual appearance of the widgets, as they are shown to the user.

According to the mapping study by Linares Vasquez et al. [11], who identified over 80 testing tools for Android apps, three categories can be derived to describe Android 2^{nd} generation testing tools, based on how the sequences of interactions are defined.

Automation Frameworks and APIs provide means to interact with the GUI of a given AUT automatically; the interaction sequences are coded in JUnit-like test methods, which are run on instrumented Android devices. The testing tools officially developed by Android, Espresso (for testing a single application at a time), and UI Automator (for testing multiple apps together with the operating system interface and capabilities) are among the most commonly used automation frameworks and APIs.

Other open-source and widely-adopted alternatives in the literature are Robolectric [12] and Robotium [13].

Record & Replay testing tools allow testers/developers to create test cases through manual executions of sequences of inputs on an instrumented device. These test sequences can be enriched with verification of specific state information of the SUT or its GUI, which are stored in repeatable test scripts.

Several of the available Record & Replay Tools are conceived as extensions of existing GUI Automation APIs, to provide another way of creating test scripts: this is the case for the Espresso Test Recorder [14], Robotium Recorder, and the Xamarin Test Recorder. Other examples of testing tools cited in the literature that leverage the record and replay approach are RERAN [15], VALERA [16], Mosaic [17], Barista [18], ODBR [19].

The most recent research in the field of Android testing has focused on *Automated Test Input Generation Techniques*, which are seen as a way of reducing the effort and cost of manually writing or recording test scripts. The creation of input sequences can be random (e.g., SAPIENZ [20], CrashsScope [21] and Stoat [22]), or model-based (e.g., MobiGUITAR [23]).

2.2. Visual (3^{rd} generation) testing tools

Third generation testing tools can automate any graphical user interface using screen captures of the individual widgets, which are used both as locators and oracles to verify the state of the AUT after several interactions. These tools are mostly agnostic to the implementation of the AUT, and they can, therefore, be used to automate any kind of application provided with a GUI – given that it is emulated on a desktop pc where the visual recognition engine can be run. Some examples of general-purpose 3rd generation GUI testing tools are SikuliX [24], EyeAutomate (evolution of JAutomate [25]), or AppliTools.

Third generation testing tools do not possess the same level of control of the assertions that can be used in JUnit-like 2nd generation test cases since they cannot verify individual properties that the GUI objects possess. Third generation assertions, instead, are based only on the visual appearance of the GUI as it is rendered on the current GUI of the app in a given state.

The validity of the 3rd generation approach to GUI testing has been proved by several studies available in the literature. As an example, case studies with the open-source SikuliX tool have been conducted at Spotify, Saab and other companies [26] [27] [28]. Other studies have proven that 3rd generation testing tools typically can guarantee easy implementation and setup, at the cost of higher expenses for maintenance [29].

To the best of our knowledge, very few studies have proposed 3rd generation testing approaches specific to the mobile domain. An exception is provided by SPAG-C [30], which obtains screen captures from an external camera that are then used to define 3rd generation SikuliX scripts.

2.3. Challenges in Android automated testing

There is a substantial unanimity in the literature about the low adoption of Automated GUI testing by Android developers. Many interview studies with practitioners have highlighted that most of the time, the preferred way of performing system testing of Android apps is to rely only on manual test cases. The low adoption of Automated GUI testing practices is not specific to the mobile domain, as several works in the literature report similar behaviour in the Web-development domain. Some of the main reasons for the lack of adoption include: the fast life cycle of software projects that prohibit automation of high-level tests, the lack of proper documentation of software tools making them costly to adopt, and the high costs for developing and maintaining test artifacts [31] [6].

On the other hand, Automated GUI testing for Android apps also suffers from a series of issues that are specific to the Android ecosystem: a very frequent amount of maintenance is needed on test cases, and the tests are also impacted by hardware and software fragmentation. Furthermore, even if in some cases they do not require high setup and development effort, GUI test cases typically exhibit a very high maintenance cost required throughout the evolution of the AUT [29].

A GUI test case can be defined *fragile* if it requires intervention when the application evolves (i.e., between two consecutive releases) due to any modification applied to the SUT [32][33]. As stated, mobile test cases are also heavily subject to fragilities since frequent changes are applied to the GUI during the app's lifespan and test cases defined with 2nd or 3rd generation automation frameworks are strictly tied to it. Many different causes can concur with the fragility in GUI test cases: in our previous works, we defined a taxonomy of 30+ types of actions on the AUT that may trigger test fragilities [34]. At a higher level, we note that it is possible to distinguish between 2nd generation-related fragilities when changes are applied to the widget definition thus causing failures in 2nd generation test cases, and 3rd generation-related fragilities, when visual modifications are performed on the pictorial GUI, and hence visual locator may not be found.

The *Fragmentation* issue includes two different concepts [35]. First, *Hardware-based* fragmentation is related to the fact that any Android app must be run on different devices, with varying hardware specifications. Hardware fragmentation has a major impact on 3rd generation (Visual) testing since also screen sizes, and pixel densities change significantly between one device and another. A valid locator or oracle for one device may therefore be unusable on a device where the same image is rendered at a different pixel density. Additionally, Android allows the developers to define different layout files for the same activities that are inflated based on the specific screen size or orientation of the device where the application is run. This type of device-related variability may impact 2nd and 3rd generation generation test cases that can be invalidated because the widgets with scripted interactions are rendered in different ways or substituted with other components. Hardware fragmentation, thus, has high costs on the practice of testing, because test cases should be re-recorded, or at least verified, on each of the devices with which the AUT must be compatible.

Second, *Software-based* fragmentation refers to the fact that several versions of the Android OS coexist, and typically apps provide compatibility to many of them. Additionally, vendors of mobile devices typically install customized versions of the Android OS. Different operating system versions typically have different graphics, hence creating the possibility of failing 3rd generation locators.

3. Motivation

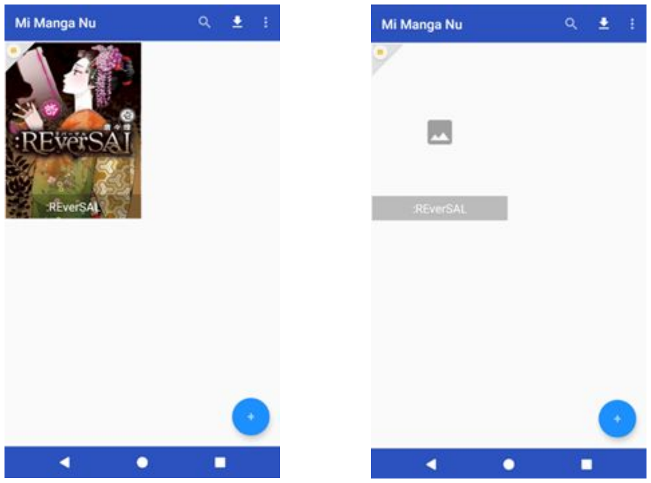
As mentioned above, combining 2nd and 3rd generation test suites can have complementary benefits, though managing them manually is often unfeasible in practice due to high associated costs [27]. Automated creation could mitigate these costs and give practitioners the value of 3rd generation scripts in a feasible manner, as shown in related work on Web-based applications [7]. For instance, the translation-based approach could be used to create visual test suites for multiple devices from a single 2nd generation test suite applicable to those devices. However, translation in the mobile domain is subjected to a couple of challenges not common to other platforms:

- More complex native interactions (e.g. hand gestures) that do not naturally translate to the mouse/keyboard inputs offered by most 3rd generation tools.

	2nd generation pass	**2nd generation failure**
3rd generation pass	2nd generation: FN 3rd generation: FN	2nd generation: TP 3rd generation: FN
3rd generation fail	2nd generation: FN 3rd generation: TP	3rd generation: TP 3rd generation: TP

Table 1: Possible combinations of 2nd generation and 3rd generation test execution in presence of faults. **TP** - True positive, **FN** - False negative.

- Fragility and fragmentation issues of moving tests between devices of different pixel-density and resolution that are not as prominent in the web- or even desktop domain.



(a) Expected final screen (b) Actual final screen

Figure 2: Example of 3rd generation true positive and 2nd generation false negative

Furthermore, it is important to note that even if a 3rd generation test suite is translated from a 2nd generation counterpart, the resulting suite will not be semantically equivalent. The reason is because of their varying means of interaction, where 2nd generation tests, as described, use widget locators whilst 3rd generation tests relies entirely on the widgets graphical appearance. These differences prohibit 2nd generation tests from verifying the visual appearance of the GUI as it is shown to the user and vice versa for 3rd generation scripts to explicitly verify the correctness of some widget properties, e.g. ids, types, etc. Thus highlighting their shortcomings, but also complementary values, in the presence of faults when used in combination. Table 1 summarizes the different theoretical outcomes of the two techniques in the presence of faults. In detail, the different outcomes can be explained as follows:

- **A fault is present but both 2nd generation and 3rd generation**

pass: In this case, both techniques fail to report a fault, i.e. a false negative result. This scenario is unlikely, and we struggle to come up with any theoretical example where this test behaviour would occur.

- **A fault is present and both 2nd generation and 3rd generation fail**: In this case, both techniques have successfully found the fault. For instance, this could occur if a component has been drastically changed or removed.

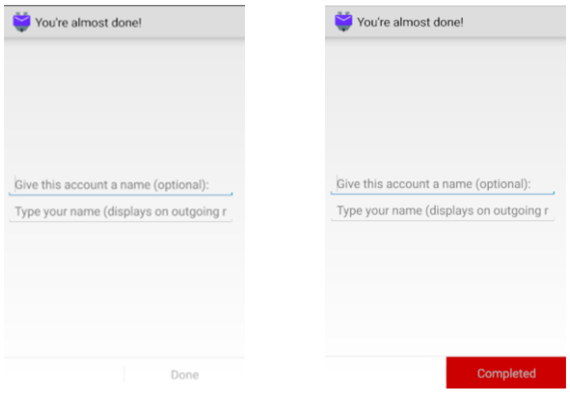
- **A fault is present and only 2nd generation fails whilst 3rd generation passes**: In this case, the 2nd generation reports a true positive whilst the 3rd generation reports a false negative. Faults of this type can be related to specific widget properties, e.g. change of ID numbers, which are not reflected in the widget's visual appearance and therefore overlooked by the 3rd generation test driver.

- **A fault is present but the 2nd generation reports a pass whilst 3rd generation fails**: In this case, the 3rd generation test case reports a true positive whilst the 2nd generation test case reports a false negative. Faults of this type generally relate to the visual appearance of the app and are not verifiable by 2nd generation test assertions. Figure 2 presents an example where sub-figure "a" reports the expected output whilst sub-figure "b" shows the actual output. The cause of the test result discrepancy could, for instance, be that the graphics library failed to load the image to the container. The 2nd generation test is only able to verify that the container is rendered, but not its visual content, and therefore passes incorrectly.

Worth noting is that, in all of these four examples, the purpose of the test must be considered when discussing the correct test behaviour. For example, for the fourth example where the 2nd generation test fails to see that the image is not loaded correctly, this is only a false negative if the intended purpose of the test was to verify that the image was properly loaded. If the intent was simply to verify the existence of a container, regardless of content, the 2nd generation test behaved correctly by passing. This example further demonstrates that the techniques have varying capabilities, which the user must be aware of, but does not diminish the contribution of this work, i.e. the cost-efficient creation of visual tests through translation. As such, TOGGLE is perceived to provide the following benefits:

- **Automated creation of visual test scripts**: This effectively enhances the existing value of available 2nd generation test cases and provides the user with automated visual testing capability at a reduced cost.

- **Reduced impact of fragmentation**: 2nd generation test scripts are device agnostic, meaning that a single suite can be used to create 3rd generation test cases for multiple devices. Thus, mitigating the test hardware fragmentation fragility [36].


You're almost done!
Give this account a name (optional):
Type your name (displays on outgoing r
Done
You're almost done!
Give this account a name (optional):
Type your name (displays on outgoing r
Completed

(a) *Done* button before graphic changes
(b) *Done* button after graphic changes

Figure 3: Sample of graphic changes applied to a widget, with layout-based properties (i.e., the ID of the button) unchanged

- **Reduced impact of graphic fragilities**: Similarly to fragmentation, translation-based creation can help in solving fragilities caused by visual changes to the GUI over time through continuous re-translation of 3rd generation tests from 2nd generation test cases. Whilst this limits the regression-testing capability for the version of the app on which the translation occurred, the benefits of automatic visual testing can still be reaped.

  Figure 3 shows an example of fragility where the text and background colour of a button has been changed. The 2nd generation test is still valid because it disregards the visual appearance, but a previously translated 3rd generation test would fail, reporting a false positive. As such, in this case, re-translation would be required for a new test that could, given that this change remains in the next version of the app, be used for visual regression testing.

4. TOGGLE

We implemented the translation-based approach in a tool, TOGGLE (Translation Of Generations of GUI testing at Low Effort). The core idea behind the proposed translator is to use the information provided by 2nd generation test scripts to create 3rd generation scripts. A first theoretical proof-of-concept of the translation approach (including the design for a backward translator, from 3rd to 2nd generation test scripts) has been presented in our previous work [9]. There we provided a high-level description of the building blocks of the architecture, and we conceptually validated the approach by modifying and translating manually 2nd generation test scripts. With the present work, we detail the actual implementation of the framework, and we evaluate it with real test cases developed for Android apps.

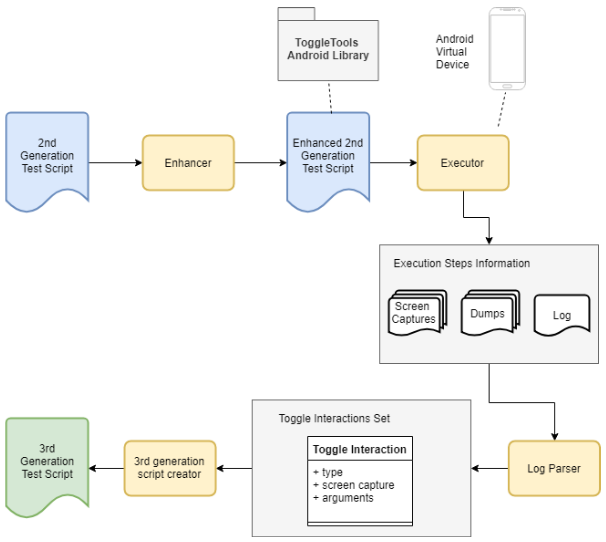

ToggleTools Android Library
Android Virtual Device
2nd Generation Test Script
Enhancer
Enhanced 2nd Generation Test Script
Executor
Execution Steps Information
Screen Captures
Dumps
Log
Toggle Interactions Set
Toggle Interaction
+ type
+ screen capture
+ arguments
Log Parser
3rd generation script creator
3rd Generation Test Script

Figure 4: Architecture of TOGGLE for translation from 2nd to 3rd generation test scripts

The translation procedure is split into two parts. First, the test scenario – a series of GUI interactions and checks – is obtained through the execution and examination of a 2nd generation test script. Second, the GUI interactions are identified, abstracted, and finally translated into the syntax of the target 3rd generation tool. Theoretically, the approach can be applied to any 2nd generation syntax, given that a module capable of parsing the specific syntax of the tool is provided.Similarly, the output 3rd generation script can be created using the syntax of different test drivers, given that a module for the creation of the scripts is developed. In our implementation, we selected Espresso as the origin 2nd generation testing tool, because it emerged from the literature as one of the most adopted tools among open-source developers [37][6]. As target 3rd generation tools, we provided translation mechanisms to both EyeAutomate and SikuliX, since they are the most cited in empirical studies about visual testing.

Figure 4 shows the building blocks of the proposed 2nd to 3rd generation test translator, along with the intermediate artefacts that are created.

The high-level architecture contains four main modules:

- **Enhancer:** it parses a 2nd generation test script, to inject function calls from the TOGGLE library into the code. This is required to extract screen captures and XML files containing the dump of the current screen hierarchy (from now on simply referred to as *dumps*);

- **Executor:** it executes the enhanced 2nd generation script on a real or emulated Android Virtual Device, checking the outcome of the test whilst saving screen captures and screen hierarchy dumps on the device memory, while logging the trace of the performed interactions;

- **Log Parser:** it parses the log saved from the executor, reconstructing the properties of each interaction and finding the exact visual locators to use in the 3rd generation test cases;

- **Third generation script creator:** it translates the intermediate and tool-agnostic sequence of interactions to the desired 3rd generation syntax.

The individual modules are detailed in the following subsections.

4.1. Enhancer

The Enhancer module, which is tool-specific, receives a 2nd generation test script as input and parses it to find the sequence of interactions that are performed against the GUI of the Android AUT.

The Enhancer module is necessary since native Android test cases are part of the application package and therefore instrumented and executed on the Android Virtual Device itself. This differs from GUI tests on web applications since it is not possible on Android to use libraries to intercept the interactions externally from the AVD: the visual captures and dump extractions have to be executed in the AVD.

The inspection of 2nd generation test cases was performed using the JavaParser library[1], identifying method calls of 2nd generation interactions. For each identified interaction, the following method calls from the TOGGLE library are added:

- *TakeScreenCapture:* The method uses the UI Automator framework [38] to take a capture of the current screen of the application. The full-screen capture is saved, as a Bitmap file, in the emulated external storage of the AVD. The screen capture is named after the test case name, followed by a progressive identifier number.

- *DumpScreen:* The method uses the UI Automator framework to extract the dump of the current screen hierarchy. The dump is an XML file, which reports all the layout properties of the widgets that are shown on the screen at a given time. The dump is saved in the emulated external storage of the AVD. Similar to the corresponding screen capture, it is named after the test case name, followed by a progressive id number.

- *LogInteraction:* The method uses the Android built-in LogCat tool, to log information about the interaction that has been performed. The logged line contains the following parameters: (i) *search_type*, i.e. the type of widget property used as a locator (e.g., "id", "text", "content-desc"); (ii) *search_keyword*, i.e. the specific value of the locator (e.g., the id "search_button"); (iii) *interaction_type*, i.e. the type of interaction performed on the widget (e.g., "click", "type-text"); (iv) *interaction_params*,

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

i.e. optional parameters that may be required to specify the interaction (e.g., the input text in case of the "type-text" interaction").

The output of the Enhancer module is hence an *enhanced* 2nd generation test script, which can still be run using the original 2nd generation tool, but that contains additional method calls able to log the nature of the gestures performed on the AUT's GUI and capture the appearance of the widgets. A sample enhancement is shown in figure 5: the dummy test case contains interaction with two widgets, using an id and textual content as locators.

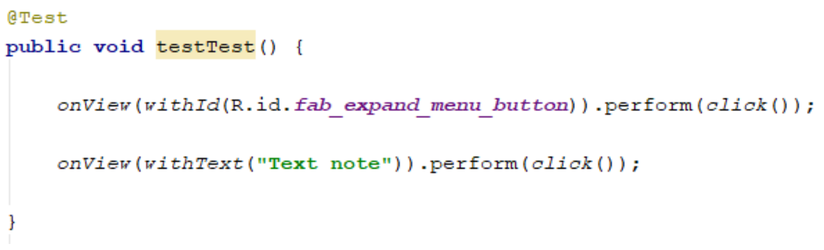
The Enhancer module is currently developed to support Espresso test cases, and is primarily tailored to identify Espresso interactions that are defined starting with an *onView* ViewInteraction, which is the primary interface - offered by the tool - to perform interactions and assertions on individual widgets of the GUI.

In the enhanced test cases, two statements are added at the beginning of each test method, in order to enable the extraction of screen captures and dump files from the emulated device. First, an Instrumentation object (that allows monitoring all the interactions between the system and the application) is obtained through a call to the *getInstrumentation* system method. Then, an instance of the UiDevice object - i.e., the UIAutomator object used to access to state information about the device - is obtained. The UiDevice instance is then used to extract the screen dump at each interaction.

The Enhancer module parses the code to find all the Espresso instructions that are supported by the tool. Each statement that corresponds to an Espresso interaction is thereby reported in the enhanced test script right after the addition of a pre-defined set of statements, including the three methods of the TOGGLE library that were described above. Each set of statements also includes obtaining the currently visible Activity, used to get the screen capture of the app. This behaviour is repeated for all lines of the original test method that contain Espresso commands; if a line does not contain any recognized Espresso interaction, it is reported in the Enhanced test file as it is, so that the layout-based test method remains executable.

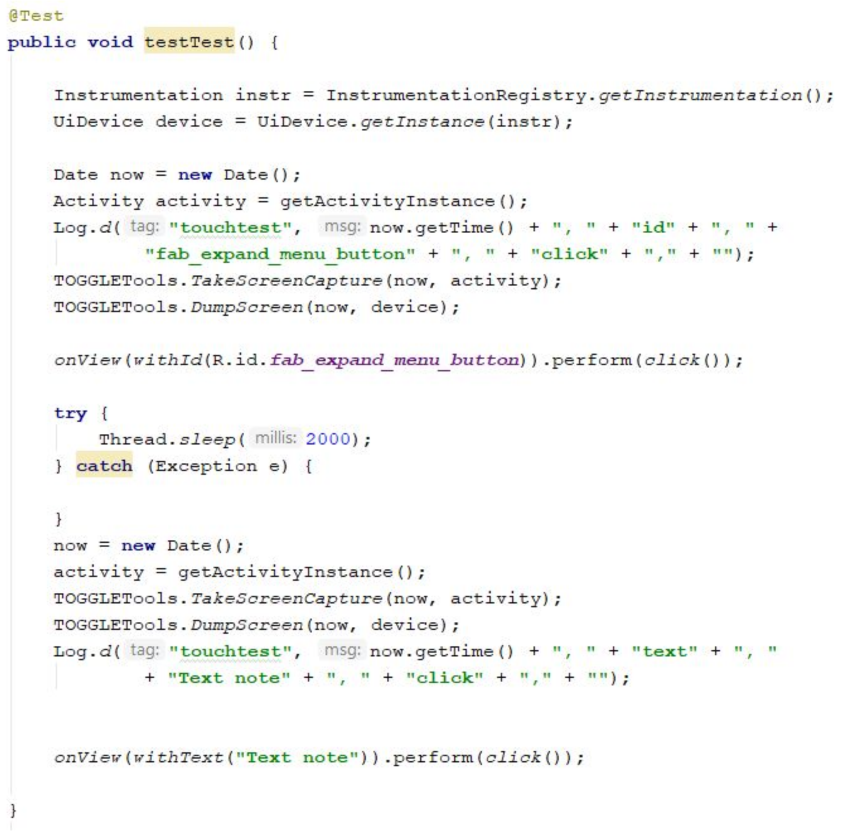
Currently, the Enhancer supports most of the interactions (each defined by a ViewAction class) that are supported by Espresso. However, some exceptions (e.g., scrolling and pressing the custom IME action buttons, and all *onData*-based commands) are still under development. The Enhancer also covers the layout-based assertions that are compatible to be translated to pure visual checks: *isDisplayed()*, which verifies that the widget is shown on screen, and *withText()*, which verifies if a text view contains a given string.

The enhanced 2nd generation test case also includes a sleep instruction between the interactions. These sleep instructions are not added to the created 3rd generation test cases, they are only present in the enhanced test cases to allow the system to have the time to obtain the screen captures and dumps. Since this sleep instruction only impacts the translation phase of the script and not the execution of the visual test scripts, we have adopted a fixed sleep time of two seconds. Such a time was observed to be sufficient for a fault-free creation



```
@Test
public void testTest() {

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

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

}
```

(a) Sample test case before the enhancement



```
@Test
public void testTest() {

    Instrumentation instr = InstrumentationRegistry.getInstrumentation();
    UiDevice device = UiDevice.getInstance(instr);

    Date now = new Date();
    Activity activity = getActivityInstance();
    Log.d( tag: "touchtest",  msg: now.getTime() + ", " + "id" + ", " +
            "fab_expand_menu_button" + ", " + "click" + "," + "");
    TOGGLETools.TakeScreenCapture(now, activity);
    TOGGLETools.DumpScreen(now, device);

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

    try {
        Thread.sleep( millis: 2000);
    } catch (Exception e) {

    }
    now = new Date();
    activity = getActivityInstance();
    TOGGLETools.TakeScreenCapture(now, activity);
    TOGGLETools.DumpScreen(now, device);
    Log.d( tag: "touchtest",  msg: now.getTime() + ", " + "text" + ", "
            + "Text note" + ", " + "click" + "," + "");


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

}
```

(b) Sample test case after the enhancement

Figure 5: A sample test case before and after the enhancement phase

of screen captures on the storage of the emulated devices.

4.2. Executor

After the 2nd generation test scripts are enhanced, the Executor module is in charge of executing them on the selected Android Virtual Device (AVD). The Executor launches the chosen AVD, installs the AUT's .apk on it (if already present, it simply calls the ADB "clear" command on it to reset its data) and executes the test cases. The device does not need to be rooted, given that the AUT is provided with the required storage permissions. Android Debug Bridge (ADB) commands are used to perform these operations. The module also ensures that the Android project is instrumented correctly and includes all the required libraries.

During the test case execution, the added methods from the TOGGLE library are called to take screen captures (.bmp images), dumps of widget information (XML files) of the screens in which the interactions are performed, and to log the information to recreate the interactions. Images, dump files, and interactions are stored in 1-to-1 correspondence since they follow the same naming convention.

The Executor also checks the outcome of the original 2nd generation test: if the test triggers any exception (failed test), the developer is notified, and the translation process is aborted. This feature is added to minimize translations of invalid tests. In fact, the fundamental prerequisite for the translation to 3rd generation test cases is that the original layout-based counterpart can go through the entire sequence of interactions without triggering any invalid state in the app.

4.3. Log parser

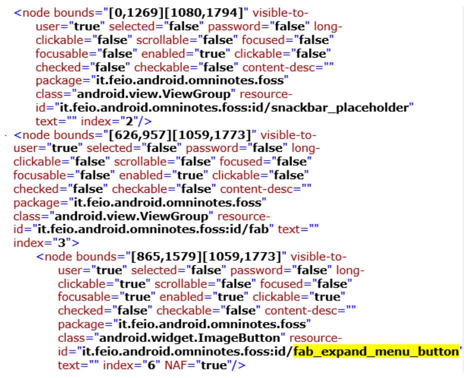
The Log Parser module is run after the Executor to capture – from the external storage of the AVD where the tests have been run – all information that is required for the translation to the 3rd generation scripts.

The LogParser module is in charge of performing the following operations for all the logged interactions:

1. It reads an interaction from the log, retrieving its parameters;
2. Using the progressive number of the interaction inside the test case, it retrieves the screen hierarchy dump which was created in the external storage at runtime;
3. Searches in the dump files for the interaction parameters (i.e., searches for a widget with the value of *search_type* equal to *search_keyword*, and extract the boundaries of the interacted widget). This step allows more precise captures of the location where the widget has been rendered at runtime. Thus, eliminating problem factors such as what device the app was launched on, the apps orientation, the position of the element in a list, etc. Hence, information that cannot be retrieved from static analysis of layout files;

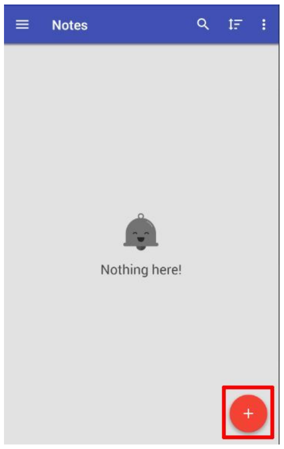
```
testAddNote, testAddNote1, id, fab_expand_menu_button, click,
testAddNote, testAddNote2, text, Text note, click
testAddNote, testAddNote3, id, detail_title, typetext, Text
testAddNote, testAddNote4, content-desc, drawer open, click,
testAddNote, testAddNote5, content-desc, drawer open, click,
testAddNote,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

(a) Log excerpt

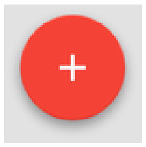


```
<node bounds="[0,1269][1080,1794]" visible-to-
    user="true" selected="false" password="false" long-
    clickable="false" scrollable="false" focused="false"
    focusable="false" enabled="true" clickable="false"
    checked="false" checkable="false" content-desc=""
    package="it.feio.android.omninotes.foss"
    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"/>
```

(b) Screen hierarchy dump with highlighted 2nd generation locator



(c) Full screen capture with highlighted bounding box for the interacted widget



(d) Visual locator for the interacted widget

Figure 6: Examples of files managed by the Log Parser module

4. Using the progressive id number of the interaction inside the test case, it retrieves the full-screen capture associated with the interaction;
5. Using the boundaries found in step 3, it cuts the bounding box of the interacted widget (i.e., the smallest rectangle that includes the image of the widget).

We report an example of the operations performed by the Log Parser in figure 6: starting from the first instruction found in the log (fig. 6a), the Log Parser identifies the exact widget inside the hierarchy dump (highlighted in fig. 6b), then uses the full screen capture of the screen to cut the exact visual locator for the widget (highlighted in figs. 6c and 6d). This visual locator, paired with the interaction info, will be the output used to create 3rd generation scripts.

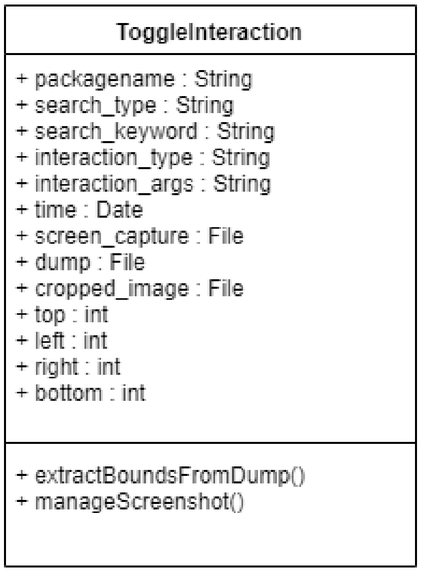


Figure 7: The TOGGLEInteraction Class

The information characterizing each set of widget properties is stored inside a TOGGLEInteraction object. This format is a completely tool-agnostic representation of each interaction with the device. The format of the TOGGLEInteraction object is shown in figure 7; the fields contain the following information: *packagename* is the name of the tested .apk, concatenated to the name of the test file, to differentiate between different test sessions; *search_type*, *search_keyword*, *interaction_type*, *interaction_args* are the field retrieved from the log line related to the interaction; *time* is the timestamp at the moment of the execution, and serves as a unique id for the interaction; *screen_capture* and *dump* are pointers to the files in external storage obtained during the execution; *cropped_image* is the visual locator for the interacted widget; *top, left, right, bottom* are the coordinates of the bounding box of the interacted widget.

Table 2: 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

4.4. Third generation Script Creator

The *3rd generation Script Creator* module depends on the Visual testing tool towards which the test case is translated. It receives as input a sequence of TOGGLEInteraction objects that are each translated into the target syntax.

In general, a 1-to-1 mapping between 2nd generation interactions to 3rd generation ones is not possible since 2nd generation interactions often act directly on the recognized views (e.g., insert a string directly inside a TextView without putting it in focus or access an item in a list which is not expanded). The development of the Script Creator module hence entails an analysis of what type of commands can be executed against the GUI of an Android app, to find the proper way of translating them into the commands featured by the 3rd generation test drivers.

This analysis requires additional effort compared to other domains where translation has been proposed. For example, for desktop and web applications mouse and keyboard operations are sufficient to replicate all possible commands. However, for mobile devices, hand gestures must also be covered.

In table 2 we report the commands that are currently supported by the TOGGLE translation tool. The table indicates if the commands are specific to Android or not and the number of visual interactions they are decomposed into. The detailed translation into 3rd generation commands in the chosen target syntaxes is provided in Appendix A. For instance, a click on a TextView is needed before sending keyboard inputs to write inside it; a swipe needs to be broken down into a button press, followed by a move command and finally a button release. Commands for pressing the buttons of Android devices (i.e., PressMenuKey, PressBack, CloseSoftKeyboard) are translated by pressing hotkeys that are captured by the Android Virtual Device.

Since the transitions in the GUI may be not immediate, depending on the app characteristics, animations, and possible race conditions with other apps running on the emulated devices, we leverage commands of the target script syntaxes to dynamically wait for the appearance of the desired widgets. These commands wait for an amount of time, that can be fixed by the programmer before the test ends up in a failure. We have set this timeout to 30 seconds, a reasonable amount of time after which the app is likely no longer changing its

Table 3: Sleep instructions added in created visual test scripts

Interaction	Sleep time
Long-click	600 ms
Swipe	200 ms
Multiple key press (e.g., Ctrl.+M)	20 ms
Replace Text	50 ms
EyeAutomate failure	5000 ms
SikuliX failure	5000 ms

GUI state. In the created test scripts, we have also added an explicit and fixed sleep instruction of one second after each interaction. This addition was made to avoid cases in which performing taps on the GUI too fast after the previous interaction could cause interactions to not be properly intercepted from the GUI engine. Finally, we have added fixed sleep times, according to the way some specific interactions - that require multiple atomic mouse and keyboard commands - are performed by the Android engine; those wait times are reported in table 3.

Another important design decision made for the Translator module was about where to insert the assertions in the created 3$^{\text{rd}}$ generation test scripts. 2$^{\text{nd}}$ generation assertions can verify varying aspects of the widgets, e.g., their textual content or parameters like their visibility on screen, whereas 3$^{\text{rd}}$ generation tools can only verify the visual appearance of widgets. Starting from the assumption that the Enhanced test is executed on a stable version of the application, we resorted to capturing visual oracles for every assertion found in 2$^{\text{nd}}$ generation code. Additionally, we added a final check of the whole screen at the end of each translated test script. This allows us to verify that the final appearance of the application, after the execution of all the test steps. A final full check is crucial to ensure that all the interactions of the test script were replayed as expected, because errors of the image recognition driver may lead 3$^{\text{rd}}$ generation tools to perform intermediate operations on wrong elements of the GUI (because of similarities with the locators used in the script) without signalling any failure. Since the EyeAutomate library suffered from false positives at the final full check, because of too many details in the images to locate, we added the possibility to tune the EyeAutomate recognition algorithm by changing the *Confirmation Threshold* parameter, which sets up the minimum similarity between the visual oracle and the rendered final screen to return a positive full check.

The output of the Script Creator is a visual test script, which can be run immediately against the app after its launch on an AVD to verify its appearance. Alternatively, the test script is added to an existing test suite for future regression testing. Hence, in addition to testing the system according to the same sequences as the 2$^{\text{nd}}$ generation test scripts, the visual scripts also verify the AUT's appearance.

At its current stage of development, TOGGLE supports translation to EyeAutomate and SikuliX scripts. The translated scripts have native formats for the two tools (i.e. plain text scripts for EyeAutomate and Python scripts for

Table 4: Translation alternatives

Name	Meaning
EA	EyeStudio Text Script
S	SikuliX Ide Python Script
EAJ	EyeAutomate Java Method
SJ	SikuliX Java Method
CES	Combined Java Method, EyeAutomate First
CSE	Combined Java Method, SikuliX First

SikuliX) that can be run by the tools' respective IDEs: EyeStudio and SikuliX IDE. However, since both tools provide Java APIs, we also equipped the 3rd generation script creator with a Java code writer. The translation of the scripts into Java test cases provides the user with richer programming capabilities that neither the native scripting languages in EyeAutomate or SikuliX provide. For instance, the created scripts could, after translation, be augmented with direct back-end interaction capability such as manipulation of the AUT's database through Java-based queries or further improved with other technical functionality. Hence, we perceive a scenario where the translator can be used to quickly get a baseline test suite that developers build upon instead of developing the baseline manually from scratch.

Additionally, the Java APIs allow translations of the 2nd generation scripts into *combined* test cases that use the Java APIs of both 3rd generation tools. These combined scripts can use the image recognition algorithms of both tools such that if one tools' image recognition fails, the script will try to perform the interaction, or the check, with the other. Two different combined, Java-based, test script types can thereby be obtained with the considered output tools: (1) with EyeAutomate interactions first, followed by SikuliX if EyeAutomate fails, and (2) with SikuliX interactions first, followed by EyeAutomate if SikuliX fails.

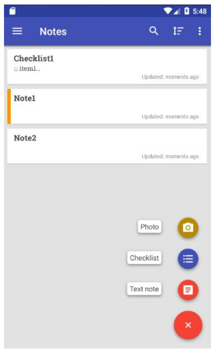
Table 4 summarizes the possible translations for 2nd generation test cases that are currently supported by the 3rd generation script creator, along with the acronyms that are used in the continuation of the paper. In the remainder of the paper, we will indicate with E the original Espresso test suite.

5. Evaluation

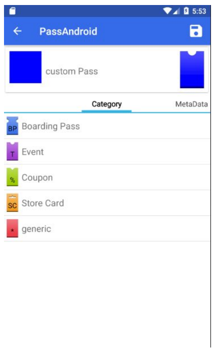
This section describes the experimental evaluation conducted on TOGGLE, the adopted procedure and its results.

5.1. Experimental Subjects

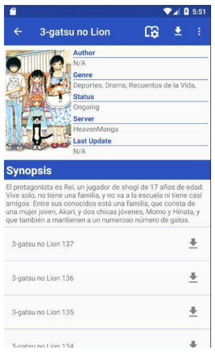
After mining GitHub repositories for Android apps that contained Espresso test cases, we found out that such repositories are scarce. Those available typically contain small-sized test suites with few test methods and trivial interactions with the GUI of the AUT. We therefore selected five different applications on which we developed Espresso test suites, on which to apply the translation-based approach for Visual test generation. One of the authors of this paper –



(a) Omni-Notes

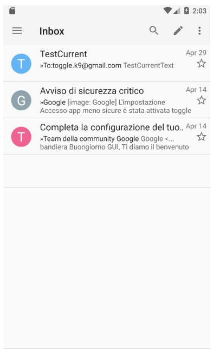


(b) PassAndroid



El protagonista es Rei, un jugador de shogi de 17 años de edad. Vive solo, no tiene una familia, y no va a la escuela ni tiene casi amigos. Entre sus conocidos está una familia, que consta de una mujer joven, Akari, y dos chicas jóvenes, Momo y Hinata, y que también a mantienen a un numeroso número de gatos.

(c) MiMangaNu

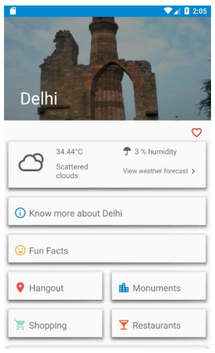


TestCurrent Apr 29
»To:toggle.k9@gmail.com TestCurrentText

Avviso di sicurezza critico Apr 14
»Google [image: Google] L'impostazione Accesso app meno sicure è stata attivata toggle

Completa la configurazione del tuo.. Apr 14
»Team della community Google Google <... bandiera Buongiorno GUI, Ti diamo il benvenuto

(d) K9-Mail



(e) TravelMate

Figure 8: Screen captures of considered applications

from now on called *Tester* – selected the mobile applications for this evaluation phase. The *Tester* was not involved in the development of the different modules of the tool. The other authors of the paper did not influence the creation of the test suites.

The following criteria guided the apps selection:

- the applications had to be native to Android;
- the application had to be open-source, and its code had to be available on GitHub;
- the application had to be a realistic Android application, i.e., not a toy application or an application with minimal features;
- the application had to have recent updates and had to follow recent guidelines for the design of Android interfaces (i.e., not implementing old design patterns).
- the application had to be released to the public or already adopted as an experimental subject in related or previous empirical studies.

The search for suitable apps was limited in time to one working day. It was also influenced by possible issues encountered when building and compiling code cloned from GitHub repositories.

We selected five applications whose screenshots are reported in fig. 8. They are:

- **K9-Mail:** a popular e-mail client, which has a long release history on the GitHub platform. The application has been used by several experimental studies in the field of mobile development and testing [39][40][41].
- **MiMangaNu:** an application for reading and organizing comics from online repositories. It served as the example of an app with possible long-running operations (the download operations of the comics) to see how they were handled with the insertion of static sleep instructions. The app is not available on the PlayStore. It has been used as an experimental subject in related literature [42][43].
- **OmniNotes:** an application for managing text notes and checklists, with possible multimedia attachments. The app is also available on F-Droid and the PlayStore. We used this application as an experimental object in one of our previous studies for the comparison of Second-generation to Visual-based approaches [44], as well as in many other studies not limited to the field of GUI testing [45][46].
- **PassAndroid:** an application for storing and managing different types of tickets through QR codes. The app has a long release history on GitHub. It is released on F-Droid and is also available for free on the PlayStore, where it has more than a million downloads. We used release 2.5.0 because of some building issues of the latest release.

Table 5: Characteristics of the selected apps (as of October 2019)

	K9-Mail	MiMangaNu	Omni-Notes	PassAndroid	TravelMate
PlayStore downloads	5,000,000+	-	100,000+	1,000,000+	1,000+
PlayStore rating	3.8	-	4.4	4.0	4.0
Number of Releases	382	72	121	100	378
GitHub Contributors	212	20	10	20	211
GitHub Stars	4,900	490	205	1,900	1,100
Tested Release	v5.708	v1.83	6.0.0 beta 7	2.5.0	5.6.2
Java LOCs	349,857	63,849	48,116	32,309	28,101
No. Activities/Fragments	60	15	13	17	35
No. Layout Files	89	14	52	19	93

Table 6: Locators used in the developed test suites

App	ID	Text	Cont. Desc.	Hint	Total
K9-Mail	68	97	21	0	186
MiMangaNu	181	99	0	0	280
OmniNotes	98	71	34	4	207
PassAndroid	131	28	9	0	168
TravelMate	42	153	17	0	212
Total	520	448	81	4	1,053

- **TravelMate:** an application for managing travels and finding information about cities. It served as an example of an app with many dynamically retrieved pictures and with the use of map activities.

General information about the size and popularity of the considered apps are reported in Table 5.

For each application, we wrote 30 test cases with the selected layout-based testing tool, Espresso. The Tester has been provided with the list of Espresso commands available in TOGGLE so that only translatable interactions were part of the developed test suites. This design choice reduces the generalizability of the experiment to any possible Espresso test suite. More details about such generalizability limitations are available in the Threats to Validity section of the paper.

The GitHub repository of PassAndroid already included some Espresso test cases. We considered those that did not contain onData ViewMatchers as part of the Tester's suite. This choice made sense from a time-saving perspective and added, to a limited extent, to the construct and external validity of the experiment. The other scenarios that led to individual test cases were instead defined by the Tester, to represent all the main features of the selected applications.

We report in Table 6 the number of locators used in each test suite. In almost half of the cases, the widgets had unique ids that could be used as locators. The second choice as a locator, in terms of frequency of occurrence, was the textual content of the widgets. Textual locators are however not as robust as id locators. They are typically more prone to change during the evolution of the app, and it is not possible to ensure their uniqueness on the screen. When the widgets do not have textual content or ids, it is possible to use Content description or

Table 7: Operations performed in the developed test suites

App	Click	Long C.	Type	Swipe	Others	Check	Total
K9-Mail	113	12	19	8	25	34	211
MiMangaNu	299	9	11	0	11	78	408
OmniNotes	110	17	35	9	13	36	220
PassAndroid	99	1	5	37	21	26	189
TravelMate	101	0	9	43	7	60	220
Total	722	39	79	97	77	234	1,248

Hints (i.e., the suggested text of a TextBox) as locators.

The test cases were built to be independent of each other, i.e., they all start from the same state of the application. As the starting point, we have selected for each application its default Main Activity. It is possible to decouple the Success Rate of different test cases by selecting a common starting point and common preconditions. This action ensures that a test case failing does not influence other ones. We designed the test cases to traverse different screens of the apps. Each test case executes from 4 to 19 interactions, ranging from simple test cases that open the menu to verify the correct rendering of specific menu voices, to more complex usage scenarios involving many transitions between activities. This variability reflects that of test cases that can be found in open-source Android projects and in the industry, where test cases can range from single interactions to 20+ different steps. The test cases were hence created to be comparable in size to industrial test cases.

It is worth noting that in one application, MiMangaNu, static sleep instructions (of 2 seconds) were added in the developed Espresso test cases. This time is necessary because the application had to connect with a database to download the comic books in the specific fragment, and the operation had to be performed before clicking on the available back button, otherwise resulting in a broken test case. These added sleep instructions in the 2nd generation test case are also added to the created 3rd generation test cases after the corresponding translated interactions.

In Table 7, we report, for each type of command provided by Espresso, the number of interactions of that type in the test suites that we created.

For the sake of readability, we included all the possible operations related to keyboard input (i.e., Type, ClearText, PressKey) under the *Type* column; in the *Others* column, we gathered operations that are not operated directly on widgets, like the PressBack and the OpenOverflowMenu. The test suites featured different distributions of commands. However, for all of them, the majority of interaction consisted of clicks.

As checks, only "IsDisplayed" assertions were inserted in the test cases. Some test cases did not feature any explicit check; in those cases, the implicit verification of the scenario was used, i.e., the test case is considered successful if it reaches its end without triggering any error state.

5.2. Research Questions and Procedure

The experimental evaluation aimed to answer the following research questions:

- **RQ1 - Tool Performance**: What is the processing time needed to translate layout-based test cases to Visual test cases with the proposed approach?

To answer RQ1, for each test case, we computed the Translation Time metric, that we define as:

$$T_{tot} = T_{en} + T_{ex} + T_{sc} \tag{1}$$

The total translation time is decomposed into three different components, each related to one of the steps needed for the translation: T_{en} is the time to perform the enhancement of the original 2nd generation test script; T_{ex} is the time to execute the enhanced script with the selected 2nd generation test driver; T_{sc} is the time for the 3rd generation script creation – including both the log parsing and the generations of the screen captures for each interaction –.

- **RQ2 - Translation Precision**: What is the proportion of interaction commands correctly translated by the tool?

To answer RQ2, for each test case, we computed the Translation Precision metric, that we define as:

$$P = \frac{I_{tr}}{N} \tag{2}$$

where I_{tr} is the number of interactions that have been correctly translated by the tool, and N is the total number of interactions that the test script encompasses. The I_{tr} metric was computed manually after an inspection of the translated test scripts. For each test case that was not translated correctly, we also identified the translation step (i.e., enhancement, execution or translation) that caused the translation error.

Since the first three AUTs on which the tool was applied –i.e., OmniNotes, PassAndroid, and MiMangaNu – were used to drive the requirement definition and initial test of the tool, we measured the Translation Precision on the last two applications we selected, namely TravelMate and K9-Mail, to avoid bias in the results.

- **RQ3 - Visual Scripts Success Rate**: What is the success rate of the visual test scripts generated through translation?

To answer RQ3, for each test case, we computed the Success Rate (SR), metric, which we define for each test script as:

$$SR = E_s/E_t, \tag{3}$$

where E_s is the number of executions ending with success, and E_t is the total number of executions. This metric thereby represents the proportion of successful executions of each test.

Additionally, using the number of successful executions of an individual test script, the tests were classified as:

- **Passing**: when all the executions end with a success (i.e., $SR = 1$);
- **Flaky**: when some executions, but not all, end with a failure (i.e., $0 < SR < 1$);
- **Failing**: when all executions end with a failure (i.e., $SR = 0$).

We assume that when all executions of a test lead to failure, and hence the test case is labelled as *Failing*, the reason of the failure must be due to an intrinsic limitation of the 3rd generation testing tool, which is incapable with finding some widget or because of an erroneous interaction with the AVD. Note that test execution is considered failed if any of its interactions fail.

We assume instead that flakiness is due to imprecision in the applied image recognition algorithm or in the recreated user interactions, which may lead to aleatory results in the executions of test cases. Another factor causing non-deterministic behaviour may be timing, where executions of the test script fail due to incorrect synchronization with the AUT's execution. This could lead to image recognition failure since the widgets may not be properly loaded at the time of the image search.

All the 30 test cases developed for each app were executed ten times. Their success rate was also averaged on the individual test suites to evaluate the ratio of passing, flaky, and failing executions.

To assess the difference among the alternative target 3rd generation tools in terms of success rate, we performed a logistic regression. In presence of categorical explanatory variables, they are converted to a set of indicator (mutually exclusive) variables that may assume values 0 or 1. Such indicator variables are defined for each level of the categorical variables; except for one of the levels that is considered the reference level (and is accounted for in the intercept). The logit regression equation we used is:

$$logit(P) = log\left(\frac{P}{1-P}\right) = \beta_0 + \sum_{t \in Tools/\{t_{ref}\}} \beta_t \cdot x_t + \sum_{a \in Apps/\{a_{ref}\}} \beta_a \cdot x_a$$

where: P is the probability of success (i.e. pass) of each individual test, β_0 is the coefficients for the reference case, β_t and β_a are the coefficients for the specific tools and apps, and x_t and x_a are the indicator variables corresponding to the specific tools and apps respectively.

We will test the statistical significance of the individual coefficients in order to decide whether to reject the null hypothesis of no difference among the tools.

While the goal is to detect differences among the tools, we include on the regression equation also the different apps to avoid the result being confounded by differences among them.

We report the average success rate of the different tests by tool and application, as well as the binomial confidence intervals using a point and range diagram. Analyses and visualizations have been carried out in a reproducible way using the R statistical package [47].

- **RQ4 - Visual Scripts Performance**: What is the performance of working visual scripts in terms of average execution time?

To answer RQ4, we measured the average execution time (T_v) of all the passing test executions. To compensate for the varying complexity of different test cases, we normalized the measured execution time by the number of interactions contained in each test case. The measured execution time depends on the sleep instructions that have been introduced for the translation of the interactions, and on possible failures of the first image recognition algorithm used in the combined third-generation test cases. The added sleep instructions are reported in table 3. These sleep instructions were added to help improve test success-rate by mitigating the mentioned synchronization challenge. The added long click delay was slightly longer than the default Android delay to detect a long click (500ms) to cope with possible lags in the execution of the application. The timeout before triggering an image recognition failure has been conformed to 5 seconds from the default values of the selected 3rd generation testing tools (respectively, 30 seconds for EyeAutomate, and 3 seconds for SikuliX), to make the execution times of the variants of the generated test scripts comparable.

Knowing the added sleep instructions, the total execution time (T_v) for a Visual test script can be decomposed according to the following formula:

$$T_v = NT_s + FT_f + \sum_{i=1}^{N} T_i, \tag{4}$$

where N is the number of interactions of the test case, T_s the sleep introduced after each interaction, F is the number of failures of the first tool used in case the combined approach was used, T_f is the timeout time to intercept the failure of the first tool, and T_i is the time for performing the $i-th$ operation. It is worth highlighting that static sleep times may be added in the original 2nd generation script, e.g. to wait for downloads or server connections. Those sleeps are not removed from the computation of the net time since they are inherent waits of the original 2nd generation test cases (i.e., they can be considered as attached to interactions performed on the GUI) and are not an overhead introduced by TOGGLE.

Based on this decomposition, the average net time per interaction in a test case can be found with the following formula:

$$T_n = \frac{T_v - (NT_s + FT_f)}{N} \tag{5}$$

The net time T_n can be deemed a more accurate estimate of the time employed by the studied algorithms for performing atomic Android commands on the emulated AVD.

We analyze the test execution time – normalized by the number of interactions – with the non-parametric permutation test. We adopted a linear model containing indicator variables – the same used in the logistic regression – and tested the significance of the coefficients corresponding to tool and application on the execution time.

- **RQ5 - Robustness to Device Fragmentation Fragility**: What is the advantage in terms of reduced fragility to device fragmentation when generating 3$^{\text{rd}}$ generation test scripts by translation?

To answer RQ5, we performed a two-fold evaluation. First, we selected the best-performing visual test suite for the Nexus 5X, in terms of success rate – measured to answer RQ3 – for all of the five applications. Then, we executed the same visual test suite on a set of 9 other devices, with varying pixel density, screen size, resolution, and the default size of the rendered AVD (see table 9 for details). We measured the success rate of such a test suite on the devices. This first result is intended to provide a quantification of the device fragmentation fragility issue for visual testing of Android apps.

Secondly, we performed a new translation of the test suites on each Android Virtual Device, separately. This step produced nine additional 3$^{\text{rd}}$ generation test suites for each application – each one provided with a specific set of device-specific screen captures – so that we could measure the average success rate of the test cases derived for the individual devices. This phase of the experiment also provides an evaluation of the device fragmentation fragility for Layout-based tests, since even Layout-based test cases originally developed for a device may not be executable on others. This issue may happen in case of adaptive Android layouts and widget disposition for different screen sizes or pixel densities.

Finally, for each target device, we compared the amount of passing (or flaky) re-translated test cases and original test cases. This comparison allows us to estimate the reduction of fragmentation-induced fragility obtained with targeted automated translation.

- **RQ6 - Robustness to Graphic Fragility**: What is the advantage in terms of the reduced fragility to pure graphic changes when generating 3$^{\text{rd}}$ generation test scripts by translation?

To answer RQ6, we applied minor modifications to the original applications. The modifications consisted in graphic changes without altering the behaviour of the widgets.

For each application, we selected 15 distinct widgets to modify; we applied different kind of graphic changes. To select the modifications to apply, we started by expanding the taxonomy of maintenance reasons for mobile test

Table 8: Types of modifications applied to the widgets

Category	Type of modification	App				
		K9-Mail	MiMangaNu	OmniNotes	PassAndroid	TravelMate
Layout	Addition	1	0	0	0	1
	Removal	2	0	0	0	1
	Position	1	1	0	0	2
Graphic	Alpha	1	0	0	0	1
	Elevation	1	0	0	0	1
	Drawable	1	8	11	9	1
	Color	1	2	2	4	2
	Rotation	2	0	0	0	1
	Size	2	0	0	0	1
	Shadow	1	0	0	0	1
Text	Alignment	1	0	0	0	1
	Style	1	2	0	0	1
	Size	1	0	0	0	2
	Color	1	1	0	0	1
	Gravity	1	0	0	0	1
	String	1	3	2	8	1
	Hint	0	0	2	2	0

scripts, that was defined in [34] by three of the authors. Within that taxonomy, only three categories of modifications can have an impact on the execution of Visual test scripts: changes in the layout, changes in the text contained by the widgets, pure graphic changes in the widget. In table 8 we report the subcategories of changes that we inferred by analyzing all the types of modifications that can be performed in the layout information of any widget, and the number of modifications applied to the five AUTs. Note that this may be higher than 15 since, in some cases, multiple variations were applied on a single widget.

The changes were not supposed to break any layout-based test suite, i.e., they did not change widget structural properties or the text when it was used as a locator in layout-based tests – due to the absence of unique identifiers or content descriptions –.

After injecting graphic changes in the apps, we performed a two-fold evaluation. First, we applied the best-performing translated test suite to the modified app, and we measured the proportion of failing and passing (or at least flaky) test cases.

Second, we re-translated the layout-based test suite for the changed application, and we measured again the proportion of failing and passing (or at least flaky) test cases. By comparing the results obtained with the original and with the re-translated test suite, it is possible to evaluate the reduction of the fragility induced by pure graphic changes.

5.3. Experimental setup

All the test cases have been run on a desktop PC with an Intel i7-8550U at 1.80GHz clock, with 16GB RAM, and Windows 10 Operating System. The development of the test suites and the execution of Espresso test cases were performed in Android Studio 3.3. The apps have been firstly launched on an

Table 9: Considered devices for the device fragmentation evaluation

Name	Size	Resolution	Density	AVD Size
Galaxy Nexus	4,65"	720x1280	xhdpi	347x617
Nexus 4	4,7"	768x1280	xhdpi	376x626
Nexus 5	4,95"	1080x1920	xxhdpi	363x645
Nexus 5X	5,2"	1080x1920	420dpi	365x649
Nexus 6	5,96"	1440x2560	560dpi	389x692
Nexus 6P	5,7"	1440x2560	560dpi	365x649
Nexus One	3,7"	480x800	hddpi	337x562
Nexus S	4,0"	480x800	hddpi	348x580
Pixel	5,0"	1080x1920	xxhdpi	352x626
Pixel XL	5,5"	1440x2560	560dpi	362x644

emulated Nexus 5X API 25 (Android 7.11) with enabled device frame and keyboard inputs. Animations were disabled on the AVD.

For multiple executions of generated test cases, single-threaded Java methods were developed; test scripts generated in the specific syntax that EyeAutomate and SikuliX have respectively been embedded in Java code and run through the use of the dedicated script runners provided by the respective APIs.

All the executions of 3rd generation test scripts were performed on a solid black background, to minimize the interference of other visual elements. No other computationally-intensive program was run concurrently with the execution of the test cases, to avoid influencing their execution time.

We needed a set of virtual devices to evaluate the graphic fragility robustness. To that purpose, the default devices offered by the Android AVD Manager were selected. The properties of the devices (size in inches of the screen, Resolution, pixel density, and size of the rendered AVD on the desktop computer) are reported in table 9. All the considered devices used x86 system images.

5.4. Experimental Results

The following subsections describe the results obtained through the designed experimental procedure, presented according to the Research Question they answer. The results provide an evaluation of the proposed approach, as well as a comparison between different 3rd generation testing tools.

In compliance with open science principles, we make available a replication package in the form of a code capsule[2].

5.4.1. RQ1 - Tool Performance

Table 10 reports the runtime (in seconds) for each step of the approach: enhancement of the test scripts (T_{en}), execution of the enhanced Espresso test scripts (T_{ex}), creation of the visual test script (T_{sc}). The table reports the absolute time for the whole test suites, and the time normalized by the number of commands for each test suite.

[2]Code capsule: `https://dx.doi.org/10.24433/CO.2149992.v1`

Table 10: Absolute and normalized execution times (in seconds) of the tool on the experimental test suites

	T_{en}		T_{ex}		T_{sc}		T_t	
Application	Total	Norm.	Total	Norm.	Total	Norm.	Total	Norm.
K9-Mail	6.49	0.03	747.3	3.54	140.90	0.67	889.7	4.2
MiMangaNu	9.31	0.02	1220.0	2.99	212.48	0.52	1441.8	3,53
Omni-Notes	8.74	0.04	638.8	2.90	115.40	0.50	763.0	3.5
PassAndroid	5.76	0.03	553.4	2.93	100.83	0.53	660.01	3.49
TravelMate	13.1	0.06	892.9	4.10	211.10	0.96	1117.1	5.12
Total	43.4	0.03	4052.4	3.25	780.7	0.62	4871.6	3.9

Most of the time was needed for the execution of the enhanced Espresso test scripts against the emulated devices. These times are much higher than those that would be measured for normal executions of Espresso test cases, because of the insertion of sleep times between each pair of instructions, and the time needed by the creation of screen captures and the extraction of screen hierarchies. The average time per interaction ranged from 2.9 seconds for Omni-Notes to 4.11 seconds for TravelMate: this higher value was likely due to the nature of the interactions, that, as shown in table 7, involved the highest number of lengthy swipe operations.

The normalized times for 3rd generation script creation were instead much lower than those for the execution of enhanced scripts, and the times for the enhancement were almost negligible if compared to the others (20 to 60 milliseconds).

Overall, the translation of test suites took between 15 to 24 minutes to complete, which is shorter compared to manual translation.

Answer to RQ1: The translation-based approach, as implemented in Toggle, was able to to perform six translations of the five test suites – 30 test cases each – in just over 81 minutes, with a normalized time of about 4 seconds per 2nd generation test interaction.

5.4.2. RQ2 - Translation Precision

We measured the Translation Precision on the last two experimental subjects we selected, namely TravelMate and K9-Mail. The results of the measured command translation rates are reported in table 11.

After translation, we noted that six test scripts required manual interventions to run successfully. These interventions simply consisted in re-capturing the screen captures for the affected scripts, which could be done with marginal time expenditure.

Five of these manual interventions were required for the TravelMate application. The reason was because of a text view that was not correctly captured by the adopted screenshot management tool since the widget was covered in the hierarchy by another widget. Additionally, one test case for the TravelMate application required a manual intervention during the enhancement phase, since

Table 11: Command translation rate results

Error	K9-Mail	TravelMate
Enhancement errors	0	0
Execution errors	2	1
Screen capture errors	0	5
Total errors	2	6
Number of interactions	211	220
Errors per interaction	0.9%	2.7%

the added 2nd generation instructions required for the translation were not compatible with the type of dialog boxes that were used in the traversed screens.

K9-Mail also required manual interventions in the enhanced versions of two test cases since the *typeTextIntoFocusedView* command was not properly logged by the tool. The reason for the error was that an Espresso interaction was performed in a way that was ignored in the translation (i.e., it was applied on a specific sub-layout of the hierarchy and not on the complete screen hierarchy as expected by the tool).

In total, there were over 200 script interactions for each of thetest suites developed for TravelMate and K9-Mail. As such, the command translation success ratio was 97.3% and 99.1%, respectively, for the two applications.

As reported in the procedure section, the first three experimental subjects (namely, MiMangaNu, OmniNotes, and PassAndroid) were used in the iterative development phases of the TOGGLE framework, to identify and correct possible translation issues. Therefore, as expected, all interactions - out of the 817 total interactions of which the three test suites are composed - were correctly translated by the tool.

Answer to RQ2: 8 out of 431 interactions (the 1.8%) – impacting 7 test cases – required manual intervention on the translated test script and/or the enhanced Espresso test scripts.

5.4.3. RQ3 - Visual Scripts Success Rate

Figure 9 reports the average translation success rates (with 95% Confidence Interval) for each test tool and application. In addition, the aggregated average per tool is reported. Note that translation success-rate is measured based on the translated scripts ability to completely execute against the experimental subject Android apps. To reference the translated 3rd generation test scripts success-rate, the diagram also includes the Espresso 2nd generation test scripts' success-rate (i.e. only execution success-rate since no translation was required), that was measured to assess the potential flakiness of the 2nd generation test cases themselves. The Espresso test cases, not surprisingly, all passed with a 100% success rate since they were developed for the experiment based on fully-working use cases of the applications, in absence of any known defect. From Figure 9 we can see that the CSE tool (Combined script with Sikuli (Java) as primary test

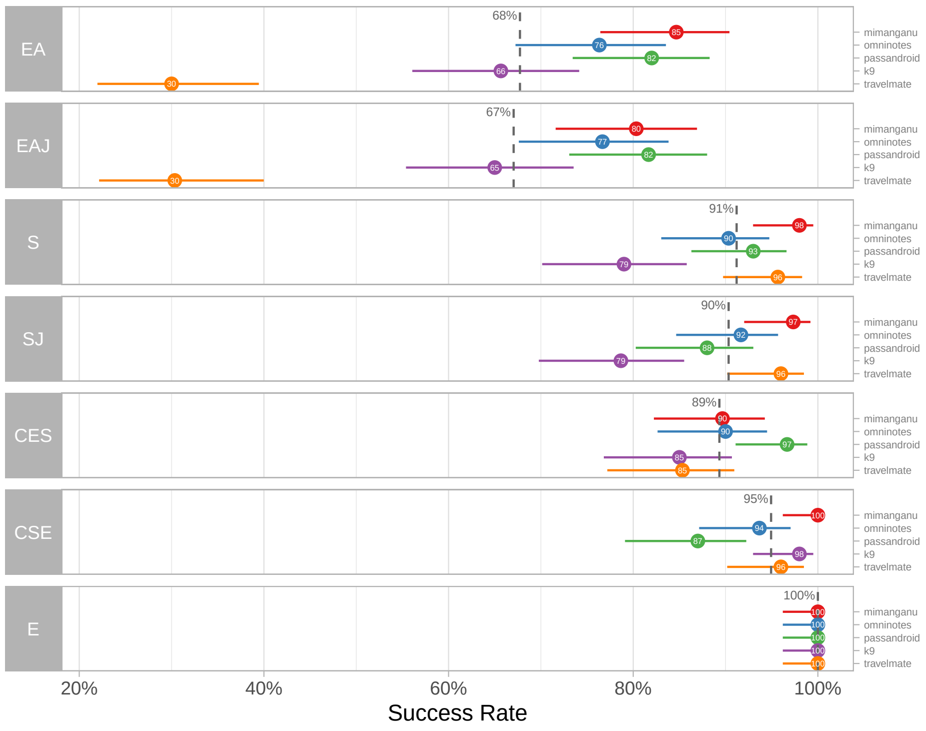


Figure 9: Average translation success rate for each test tool and app plotted with 95% confidence intervals. **EA** - EyeAutomate (Native), **EAJ** - EyeAutomate (Java), **S** - Sikuli (Native), **SJ** - Sikuli (Java), **CES** - Combined (EyeAutomate (Java) with Sikuli (Java) as backup), **CSE** - Combined (Sikuli (Java) with EyeAutomate (Java) as backup), **E** - Espresso.

driver and with EyeAutomate (Java) as backup) exhibits the highest average success rate (95%) and the EAJ (Combined script with EyeAutonate (Java) as primary test driver and with Sikuli (Java) as backup) the lowest (67%).

Table 12 reports the results of the logistic regression. We observe that all 3rd generation tools, except EAJ exhibit a significant difference (all p-values $< 10^{-3}$) in terms of success rate from the reference tool, i.e. EAJ. Moreover we can observe a significant difference among the apps.

The EyeAutomate tool, both when running with the specific plain text syntax through the Script Runner or in Java Code through the usage of its APIs, was the least successful, with average success rates of 68% and 67% respectively. Average success rates for the EyeAutomate test cases ranged from around 30%, for the TravelMate app, up to around 85%, for MiMangaNu.

The average success rate for SikuliX test cases was higher than 90%. Breaking down the results by App, we observe peaks near 98% for the MiMangaNu app. As we can deduce by looking at the confidence interval, no significant difference could be found between the average success rate of scripted versions of the test scripts and Java counterparts, for both SikuliX and EyeAutomate.

Table 12: Logistic regression result for Success Rate (the reference level is consists in the tool EA and the app MiMangaNu

β	Estimate	CI	Std. Error	p-value
β_0	1.562	(1.368 , 1.756)	0.099	< 0.001
ToolEAJ	-0.040	(-0.198 , 0.119)	0.081	0.624
ToolS	1.675	(1.461 , 1.888)	0.109	< 0.001
ToolSJ	1.568	(1.361 , 1.776)	0.106	< 0.001
ToolCES	1.456	(1.254 , 1.657)	0.103	< 0.001
ToolCSE	2.279	(2.020 , 2.538)	0.132	< 0.001
Appomninotes	-0.588	(-0.810 , -0.365)	0.114	< 0.001
Apppassandroid	-0.429	(-0.657 , -0.202)	0.116	< 0.001
Appk9	-1.208	(-1.419 , -0.998)	0.107	< 0.001
Apptravelmate	-1.599	(-1.806 , -1.393)	0.106	< 0.001

Similar average success rates were obtained with the usage of combined output techniques.

Overall the combination of SikuliX first and EyeAutomate second (CSE) was significantly better (comparing CIs) than CES, SJ, and S that showed no statistically significant difference among themselves, and in turn significantly better results than EA and EAJ. A sort of exception is PassAndroid, for which the best tool was CES. This outlying result was mainly due to more robust test execution behaviour of the EyeAutomate tool when swipe operations are involved, better detailed later.

The breakdown of the proportion of Passing, Flaky and Failing Tests, measured for the six sets of 3rd generation scripts and divided by app are reported in fig. 10. We can observe that for all the five applications, a high percentage of EyeAutomate test cases (both with the test scripts and through the Java APIs) failed in all executions. This percentage reaches 70% for TravelMate. On the other hand, test cases written with SikuliX showed no failing test cases for MiMangaNu and a maximum 17% of failing test cases for K9-Mail.

The usage of combined 3rd generation test cases led to even better results, thanks to the usage of a backup visual tool when a recognition with the first tool failed. While the combination with EyeAutomate as the primary tool had a residual amount of failing test cases, the combination with SikuliX as primary tool proved to have the lowest number of failing test cases overall: just a single one for PassAndroid and TravelMate.

In addition to reporting the success rate distributions, we also analyzed – based on the different results – the individual test cases, to understand the reasons that led some tools and test cases to fail.

For instance, the EyeAutomate visual recognition library was unable to find visual components like the Navigation Drawer icon (consisting of only three white lines on a blue background, see figure 11), and the More Options icon (consisting of three small dots, see figure 12)[3]. Some flakiness in SikuliX test

[3]Discussions with the tool's developers revealed that the reason for these failures was likely because the tool's image recognition algorithm requires a certain amount of information (e.g.,

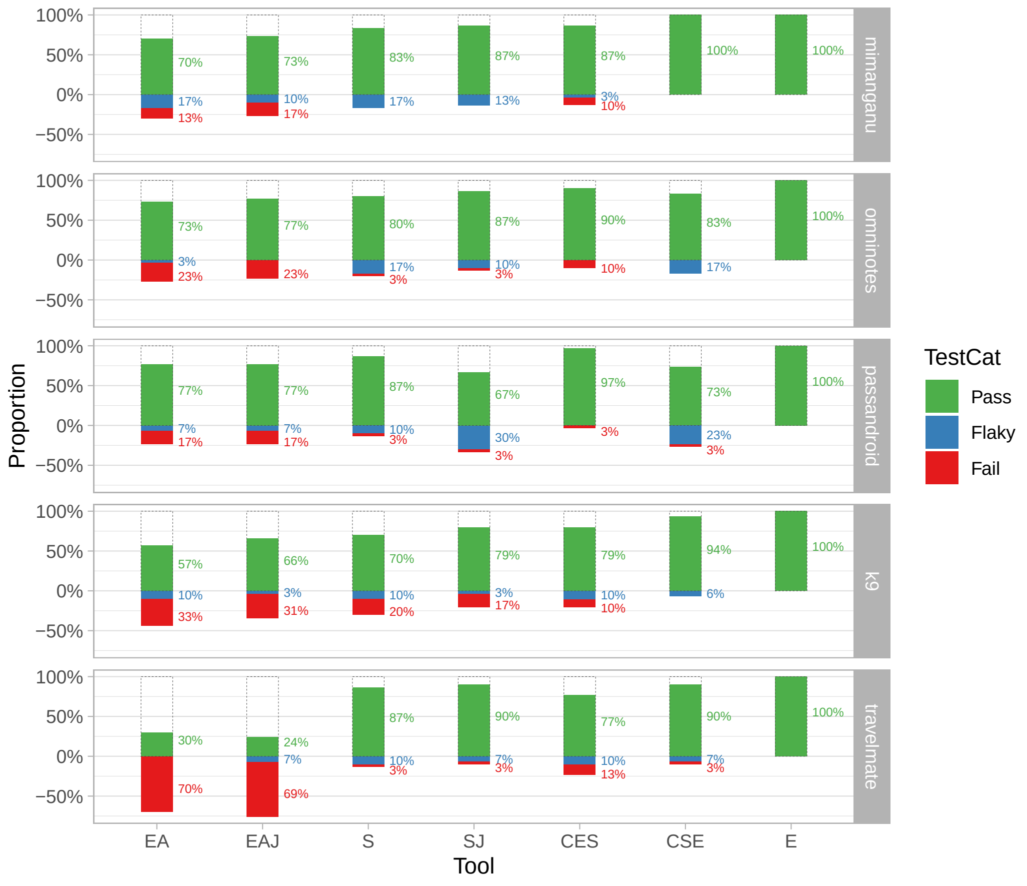


Figure 10: Proportion of passing, flaky and failing translated test cases

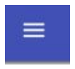


Figure 11: Navigation Drawer button (screen capture taken from the OmniNotes app)

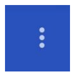


Figure 12: More Options button (screen capture taken from the MiMangaNu app)

Table 13: Average number of backups for combined methods

App	CES	CSE
K9-Mail	0.81	0.20
MiMangaNu	0.64	0.01
Omni-Notes	0.27	0.10
PassAndroid	0.17	0.14
TravelMate	0.47	0.12
Overall 1	0.43	0.08

cases was connected to the need for swipe operations, which were less precisely reproduced[4].

The described failures showcase the varying capabilities of different image recognition algorithms and also a secondary benefit of translation. Hence, translation can not just be used to transfer one generation of GUI tests to another, but also allows translation to different technologies, or combinations of technologies, to best fit a certain context or purpose.

Hence, the combination of the tools improves the overall success rate for all apps. The CES combination had a residual number of failing test cases even when all executions were passing with CSE. Those remaining failing test cases may be justified with situations in which EyeAutomate executes an operation on a wrong locator (i.e., a false positive of the image recognition engine), hence deviating the test case from its correct execution. In contrast, when an EyeAutomate test gets stuck for not recognizing a widget, using the image recognition algorithm of SikuliX as a backup allows “runtime repair” of the test case without moving to the wrong states of the GUI.

Table 13 reports the average number of times the ”backup” tool was used in the test cases. The overall values confirm that the SikuliX tool proved more robust, being used more often as a backup of a failing EyeAutomate locator than the vice-versa.

Answer to RQ3: None of the 3rd generation scripts achieved the same success rate as Espresso test cases for all the three test suites considered for our evaluation. The experiment proved, however, that very high success rates (with peaks of 100%) can be obtained with visual test scripts created through translation. The combination of multiple image recognition algorithms, with one used as a backup for the other, proved to be a valid enhancement for the success rate of translated tests.

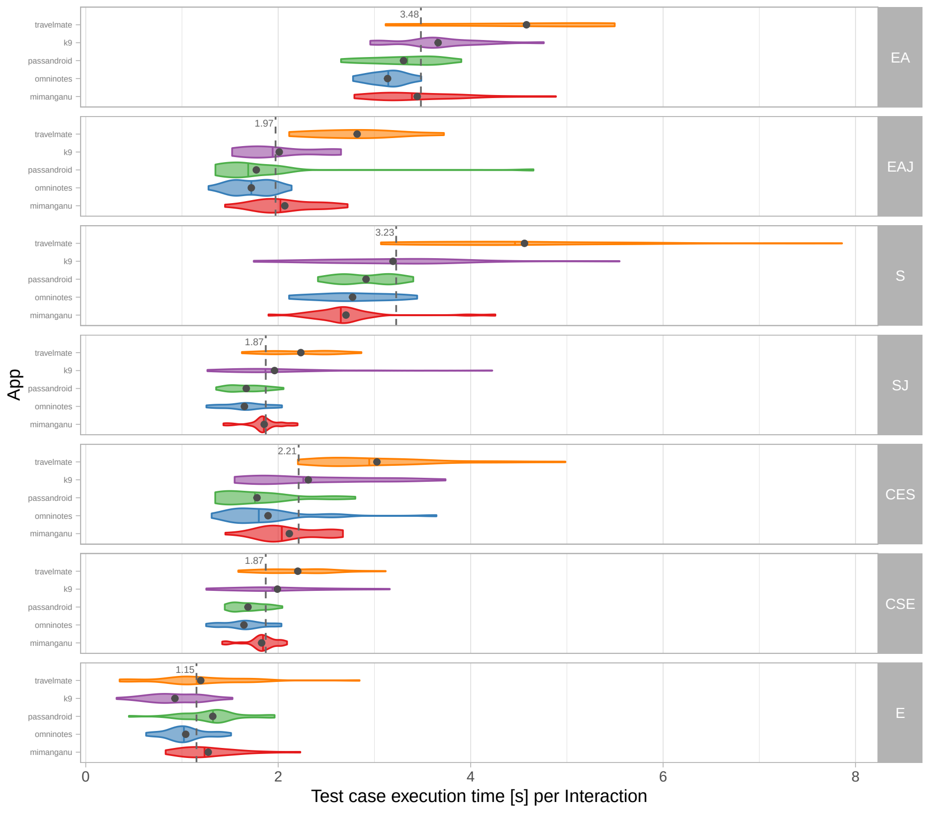


Figure 13: Distribution of execution time, normalized by number of interactions, by tool and app

Table 14: Linear model of time per interaction vs. Tool and App and test result (the intercept corresponds to the reference level EA:MiMangaNu).

Coefficient	Estimate	p-value
(Intercept)	3.472	< 0.001
Tool-EAJ	-1.511	< 0.001
Tool-S	-0.345	< 0.001
Tool-SJ	-1.701	< 0.001
Tool-CES	-1.352	< 0.001
Tool-CSE	-1.699	< 0.001
Tool-E	-2.421	< 0.001
App-omninotes	-0.196	< 0.001
App-passandroid	-0.114	< 0.001
App-k9	0.109	< 0.001
App-travelmate	0.702	< 0.001

5.4.4. RQ4 - Visual Scripts Performance

Figure 13 presents the test case execution time for each tool and app, normalized by the number of interactions performed. Only the passing test case executions were taken into consideration for the computation. Checks (either of individual widgets or the full screen) were counted as interactions since the time required by the image recognition algorithm to find a match is equivalent regardless if the purpose is to identify a position for interaction or simply to find if a widget is present. Once more, Espresso has been added as a benchmark to see how the other tools compare. The number of interactions performed in Espresso test cases was the same as in the translated 3rd generation ones, except the final full check of the app screen (i.e. the assertion) that was not present in developed Espresso test cases.

Table 14 reports the coefficients for the linear regression of the time per interaction vs. the indicator variables corresponding to the different tools and apps. The non-parametric permutation test on the linear model coefficients shows a significant difference between measured average time per interaction depending on tool (all $p < 10^{-16}$) and a significant effect of the application (all $p < 10^{-16}$). In other words, the results say that:

1. changing the target tool of the translated scripts is sufficient to provide a significant change in the measured time per interaction, due to varying image recognition algorithms adopted;
2. changing the AUT leads to a significant change in the measured time per interaction, a reasonable result since different AUTs may need different sets of actions and varying delays.

an image of large enough size or advanced enough pattern) to accept the image as a match. The three lines or dots did not fulfil these criteria and were therefore ignored.

[4]An analysis of the SikuliX code suggested that additional overhead is added by the SikuliX methods to mimic a smoother, human-like interaction with the AUT. This overhead may cause a slower movement of the Android widgets, that are moved back to the original position if the swipe movement is too slow.

Table 15: Average time and average net time per interaction, per tool (seconds)

Tool	Time per int.	Net time per int.
EA	3.48	2.48
S	3.23	2.23
EJ	1.97	0.97
SJ	1.87	0.87
CES	2.21	0.97
CSE	1.87	0.82

The magnitude of average time variation induced by change of the tool is one order of magnitude larger than switching to a different AUT.

Espresso guaranteed a lower average execution time per interaction. The main reason for this is the tool's use of properties that has inherently higher performance due to less required calculations than the image recognition approach. Additionally, Espresso, being integrated into the Android framework, can filter the intents for Activity switching and automatically wait for the exact time for an Activity or widget to appear on the screen, thus minimizing waiting times. The higher execution time of 3^{rd} generation tools is a finding that has been reported by many works in the literature [48][10], and the results reported in this paper are in line with manuscripts comparing the performance between the two technologies. Similarly, the comparison between the execution time of different 3^{rd} generation tools is a supporting contribution of this study.

The difference in average time per interaction caused by the different apps can be explained by the fact that the patterns of interactions with the five applications are different, e.g., PassAndroid and TravelMate required more longer swipe operations than the other AUTs.

The fastest tools after Espresso were the Java version of SikuliX, and EyeAutomate's Java API. Hence, an interesting observation is that, both tools had lower performance when tests were written in the tool's specific syntax than in their respective Java APIs. This may be explained by the fact that the test cases were run inside a Java environment, instantiating script runners provided by the respective libraries. An alternative explanation is that the script tools' implementations caused additional overhead that is not present in when the bare-bone image recognition libraries are used.

As expected, the combined test versions had a bit worse performance than any of the tools individually. The reason is TOGGLE's approach of creating tests that always try with one tool first, and only if it fails, after a set time (of 5 seconds), uses the second tool. Both combined solutions had, however, a better performance than the scripts developed in the tool-specific syntaxes.

Table 15 shows a comparison between the average interaction time for all the tools, and the relative net interaction times (obtained by removing sleep and backup times that had been inserted in the test cases).

We measured a relevant difference also between the net interaction time required by the scripted versions of the tests compared to the test suites leveraging the Java APIs of the two adopted testing tools. The difference is of more than one second per interaction for both SikuliX and EyeAutomate. No substantial difference in terms of net interaction time, on the other hand, was found between the test suites written in Java, with CSE (Combined Sikuli-Eyeautomate) being the fastest and CES and EJ (Combined EyeAutomate-Sikuli and the Java API of EyeAutomate) the slowest.

Answer to RQ4: The 2nd generation approach, as expected, has significantly lower execution time compared to any of the single or combined 3rd generation solutions. We also measured a significant difference between the average time per interaction measured with the six considered 3rd generation testing tools, with the Java version of Sikuli being the fastest.

5.4.5. RQ5 - Robustness to Device Fragmentation

To evaluate the fragmentation fragility reduction, we utilized the combined Sikuli-Eyeautomate (CSE) test suites, obtained from the previous experiments, since they had the best overall behaviour in terms of success rate for all AUTs, on the Nexus 5X.

The dumbbell plot in fig. 14 shows the success rates of the visual tests originally captured and converted on the Nexus 5X and executed in nine other different devices (bullet), versus the success rate of the suite – automatically – re-captured on the very same devices (triangle).

We observe that the test cases translated on the Nexus 5X (bullets in fig. 14) were almost completely portable to the Nexus 6P and Pixel XL devices, likely because of the similar size of the pictorial rendering of the device on-screen. On the other hand, most likely due to rendering differences of varying pixel density, the test suite was not fully portable to the Nexus 5, even though it shared the screen size with the Nexus 5X. The portability was also limited on Nexus 6 and Pixel, which was caused by minor changes in the rendering of the buttons. For devices with smaller screens (Galaxy Nexus, Nexus 4, Nexus One, Nexus S), the tests could rarely be ported due to the very different sizes of the rendered widgets. On average, on all devices, only 31.6% of visual test cases were portable (less than 10% for five devices out of nine).

These results clearly demonstrate that the negative impact of Device Fragmentation on Visual tests is quite high for Android applications when the screen size and the pixel density of the target device are different from those of the device on which the test suite has been captured.

On the other hand, looking at the success rate of the re-captured test suites (triangles in fig. 14) , the vast majority of the test cases that were translated to specific devices, starting from a common layout-based counterpart, were successful (at most flaky). Two devices (the Nexus S and Nexus One) exhibited the lowest percentage of working translated test cases. This was caused by the fact that several Espresso test cases (3 for OmniNotes, 2 for PassAndroid, one for MiMangaNu, eight for K9-mail) were not executable on those devices. Due to their smaller screen size, different layouts were rendered, with widgets that were not displayed to the users. Whilst this was a hindering result for the experiment, it also showed a benefit of translation, since the3rd generation tests

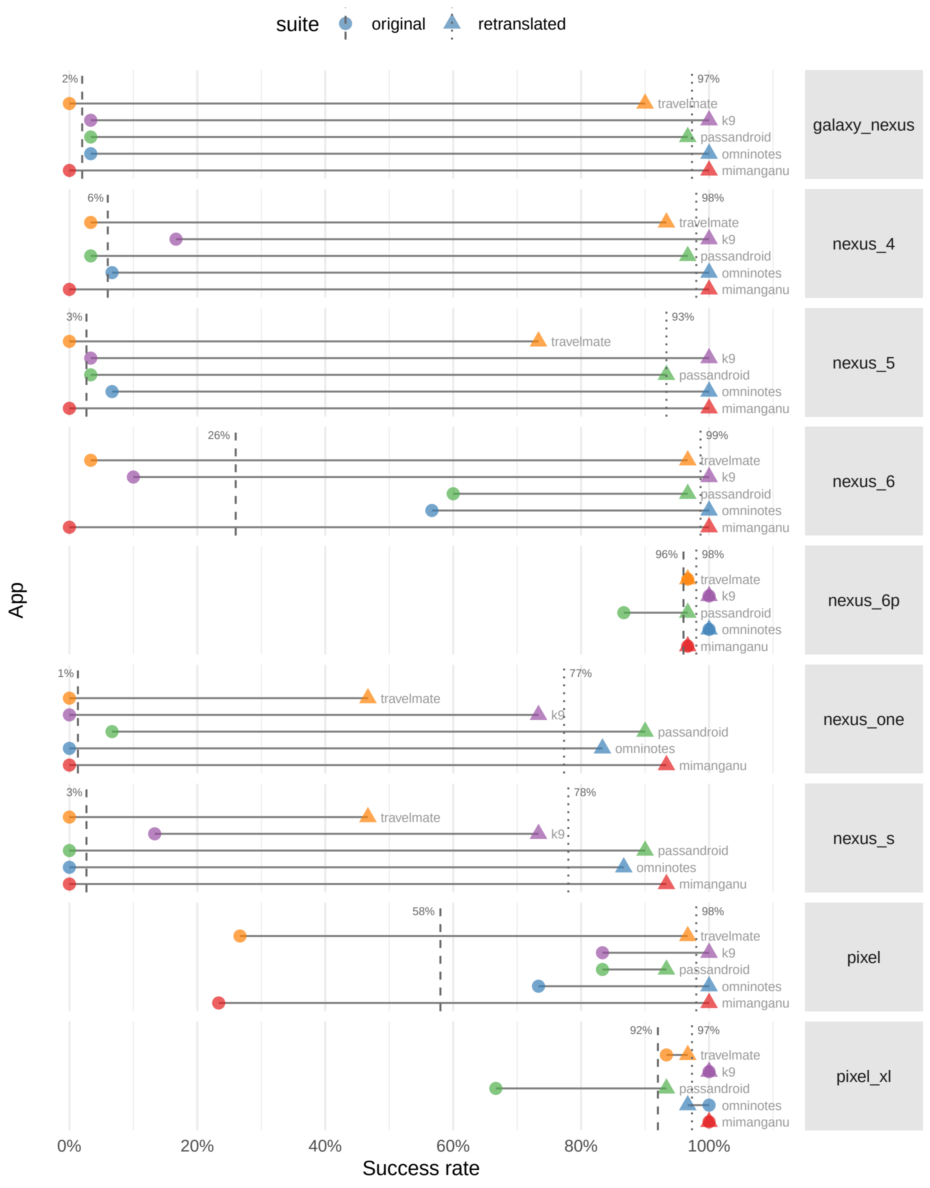


Figure 14: Change in visual test success rate between suite originally captured on different device and re-captured on same device.

would fail due to layout issues. In fact, some of these faults could be classified as being detrimental to app usability, i.e. tests of a non-functional attribute of the AUTs.

Also, the options button was not shown because a physical button – then removed from Android devices – was used to that purpose. In these cases, the layout-based test cases themselves were fragile to device diversity. Thus they could not be used to create valid image recognition-based counterparts. For all other devices, 90.0% or more of the translated test cases were passing or flaky.

The described results suggest that it is possible to adapt existing layout-based test suites to varying devices with minimal effort, to be spent in modifying the residual visual locators or oracles that cause false negatives in the translated 3rd generation test cases.

Answer to RQ5: Only 31.6% of the visual test cases, on average, were portable to other devices. The use of a translation-based approach that creates test cases on different devices starting from a common set of layout-based tests achieved a better result with 93.3% portability of a sample of 150 test cases, developed for five applications, over nine devices with varying characteristics.

5.4.6. RQ6 - Robustness to Graphic Fragility

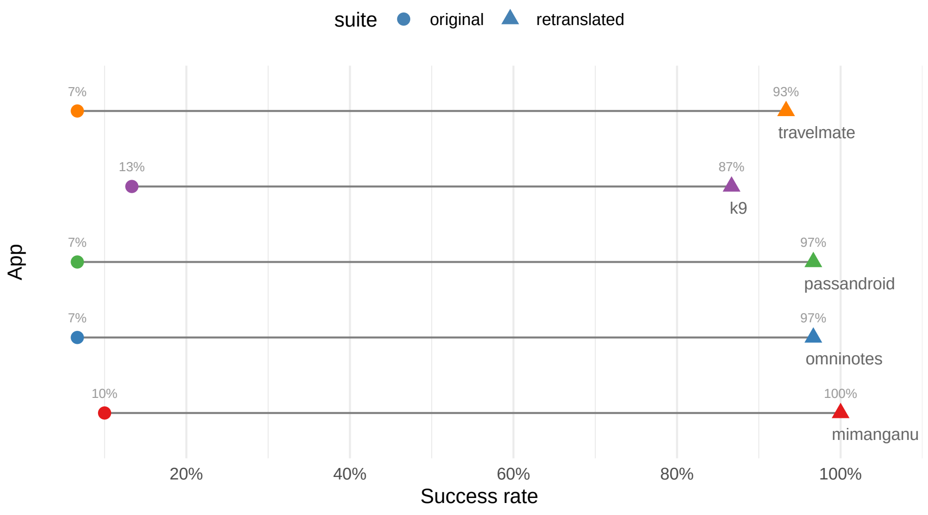


Figure 15: Percentage of passing (or flaky) test cases for the original test suites, and for the test suites re-translated after graphic modifications were applied to the apps

The dumbbell plot in figure 15 report the results of the evaluation that we performed to measure the robustness to Graphic Fragility of the translation-based approach. We injected graphic modifications to 15 separated widgets for each of the five software objects. In this case, we utilized the CSE output combination of the TOGGLE tool as the test suite for our experiment.

In the plot, we report the percentage of passing or at least flaky test cases for each app, for the test suite translated before modifications were injected (bullet), and for the one re-translated after the modifications were applied to the app (triangle). The discussed results can be considered as a proxy to evaluate the benefits of the application of a translation-based approach to cope with graphic maintenance of existing AUTs.

It can be seen that the graphic changes invalidated the vast majority of the original 3rd generation test cases for all the considered apps. The number of original test cases that still passed on the modified AUTs ranged from 2 to 4 (for K9-Mail).

On the other hand, when the test cases were re-translated, the number of passing ones ranged from 26 (for K9-Mail) to 30 (for MiMangaNu).

Some test cases did not pass even after the translation: for instance, the changes in OmniNotes involved a modification in a small TextView that likely was not recognized after the modification by both the 3rd generation drivers. For K9-Mail, three test cases were not re-translatable because of the Espresso tool itself not working properly on rotated views.

However, the effort of repairing those test cases (for instance, by selecting a bigger portion of the screen instead of just the bounding box of the specific widget, or by recreating an interaction with a rotated widget) can be deemed minimal if compared to the manual re-capture of all the changed widgets, and the manual fixing of all the test cases using them.

Answer to RQ6: The automated re-translation of test cases provided a reduction of around 90% of the occurrence of graphic change fragility. While just 13 visual test cases out of 150 could still be used on the changed GUIs, the re-translation with TOGGLE was able to repair 128 test cases, for a total of 142 working test cases out of 150.

6. Discussion

The experiments we conducted have highlighted the feasibility and the benefits of a translation-based approach from 2nd generation to 3rd generation in the mobile testing domain.

The migration of a layout-based test suite to a visual one lowers the need for costly manual operations – for both creation and maintenance – in any application domain. These costs are particularly high for mobile applications, where changes in the GUIs are frequent and where fragmentation issues (related to both graphical modifications and device change) have a significant impact.

We implemented the proposed approach in a tool named TOGGLE. Although the tool covers a subset of interactions available in Espresso, it is capable of translating the most commonly used ones – w.r.t. test suites developed with such tool – and this translation proved to be fast and correct for nearly all interactions.

A thorough assessment of any tool requires the evaluation of the usefulness of its output. Therefore we evaluated two usage scenarios: (i) re-translation of the same test suite in case of fragility induced by device fragmentation; (ii) re-translation of the same test suite in case of graphical fragility. We observed the portability of test suites to different devices was enhanced by this approach; this would likely lead to reduced maintenance costs when the graphical features of the AUT are changed.

As such, the results provide a proof of concept and indicate that the users of such a translation-based approach can get the benefits of visual testing, with a significant reduction of the cost for capturing the right oracles and locators.

It is important to notice that the current approach assumes that the AUT is not faulty, i.e., that the 3rd generation test cases are obtained on a version of the application that has no regressions. If defects are encountered at the time of translation, wrong captures may be obtained for both visual locators and oracles, leading to erroneous sequences of interactions in the resulting visual test scripts, that would require manual effort from the testers to be repaired. We observe that such drawbacks are similar to those affecting model-based testing, i.e. the automated inference of models of the GUI from the AUT. This latter technique is significantly cheaper than the manual creation of models, however, it may produce faults in the generated models, hence requiring validation and additional information that has to be provided manually [49][50].

6.1. Practical implications

It is important to emphasize the consequences of our findings in the context of practical development:

- *Suitability for automation*: the translation process is entirely automated; manual intervention in case of wrong translations, that in our experience affected less than 2% of test cases. We also need to stress that ours is a proof-of-concept tool, not and industrial-grade instrument.
- *Translation efficiency*: the tools is able to translate 2$^{\text{nd}}$ generation test cases into 3$^{\text{rd}}$ generation ones at a pace of one every four seconds.
- *Test dependability*: the resulting 3$^{\text{rd}}$ generation test cases were less dependable than the 2$^{\text{nd}}$ generation counterparts. The combination of different image recognition algorithms improved that, but still the proportion of passing tests ranged between 73% to 100%. As a disclaimer, we observe that these are limitations inherent in 3$^{\text{rd}}$ generation tools and are not specific to our approach. Therefore future improvements in this category of tools would trigger improvement in our approach too.
- *Test execution*: the execution of the translated 3$^{\text{rd}}$ generation test cases took roughly 60% more time than the 2$^{\text{nd}}$ generation ones. As for dependability, here the approach is limited by inherent characteristics of the 3$^{\text{rd}}$ generation tools.
- *Device fragmentation reduction*: our approach was able to raise reproducibility of tests across different devices from 32% to 93%. This is not a definitive solution, though it represents a powerful mitigation.

- *Graphical change fragility*: the re-translated test cases showed and enhanced average reproducibility of 95% versus the original 9%. In practice our approach dramatically improved the resilience of 3rd generation tests then purely graphical changes are applied to applications.

6.2. Current limitations and open issues

As exposed in the tool's implementation details, the TOGGLE tool currently only works for widgets that can be interacted with through calls to the *onView* family of methods. For this reason, the tool is unable to execute commands directly on elements of dynamically populated structures, like RecyclerViews and GridViews, with custom layout descriptors for the individual elements. However, if those elements have textual content, it can still be used as a layout-based locator to be translated into a visual one (with possible movements in the user interface with swipe operations if the element is outside the current screen of the app).

Among all the possible ViewActions that apply to Android Widgets, the tool still does not feature an automated translation for the ScrollTo interaction. The difficulties in translating this operation are mainly related to the calibration of the slow scrolling that is needed to find elements inside a scrollable Adapter based on its appearance. If the scrolling happens too fast, it may go past the sought widget, and if it is too slow, it will negatively affect the scripts' performance. We are currently seeking ways to implement this exploration of scrollable elements of the GUI to avoid adding excessive overhead to the generated visual test scripts as well as guaranteeing sufficient dependability.

Also, the PressIMEActionKey interaction is still not implemented because of the inability to take screenshots of the Virtual Keyboard with the instrumentation that we are currently using. We are aiming to provide coverage of this functionality by implementing clicks on a specific part of the emulated device known to host the IMEActionButton in the Android default virtual keyboard (right-bottom corner).

Furthermore, the tool currently has no support for finding visual elements that have the same appearance as others, i.e., a generated Visual test is likely to report a false negative result if multiple widgets in the interface share the same visual appearance. This situation is quite common for Android apps, e.g., for menus of Radio Buttons, and therefore is classified as a major challenge for the approach. We are aiming at implementing the management of elements with the same appearance by maintaining a screen capture of the whole GUI for any interaction, and by finding coordinates of the widgets to be interacted with inside the whole screen. A similar approach has been proven beneficial in the literature [51].

In any case, the current limitations of the tool can result in the need for minor manual adjustments on the generated test suites. This effort in fixing oracles and locators is lower than that needed for the full manual re-capture of a visual test suite.

7. Related Work

The proposed approach adds to studies available in the literature, that conceptualize the possible benefits of a combined approach of 2nd and 3rd generation testing tools [2]. The results of those empirical evaluations show that the 2nd generation approach interacts and asserts the GUI model, leading to more false-negatives than 3rd generation approach for acceptance test; on the other hand, the 3rd generation approach, which mimics a real user's interaction with the GUI, reports more false positives than the 2nd generation approach for system testing. The different behaviour and the different type of information that is verified against the actual state of the application suggest that the techniques should be adopted in combination for better test performance. 2nd and 3rd generation testing tools have also been compared in terms of learnability, quality, and robustness of the developed test suites, as perceived by practitioners [44].

The present work is also related to existing literature that aims at combining layout-based and visual testing, that evaluates the benefits and drawbacks of both techniques, or that proposes novel methodologies to generate more robust and portable visual locators.

7.1. Translation-based approaches

A translation-based approach similar to that used by TOGGLE has been already proposed by Leotta et al. in the field of Web-Application testing, where DOM-based 2nd generation test cases (developed with Selenium WebDriver) were translated to 3rd generation test cases (written with Sikuli) [7][52]. The reported evaluation of the tool highlighted the enhanced maintainability and ease of re-creation of 3rd generation test cases, compared to the original 2nd generation ones from which they were obtained.

The said approach is based on the translation of DOM-based test cases, that can be generalized to any web application, even web-based or hybrid mobile applications. The approach we propose is instead specifically tailored to Android apps since it is based on layout-based test cases which use native properties of Android apps as locators (e.g., unique ids or content descriptions).

Our approach also covers a higher number of interactions with the SUT than PESTO, which instead only covers click, type and check instructions. This property is due to the higher number of instructions featured by the platform-specific tools considered as the source for the translation. The TOGGLE tool needed to be designed to manage commands that cannot be translated directly to atomic 3rd generation instructions, e.g., scroll and swipe operations.

Compared with PESTO, our approach does not consider the possibility of interacting with multiple elements with the same on-screen appearance, even though this limitation can be solved by future developments of the project.

On the other hand, while the test cases generated with PESTO required some (even though minimal) manual adaptation of the generated test code, our tool does not require any manual adaptation of the generated visual test scripts.

The difference between the architectures required by PESTO and our tool underlines how – albeit being similar in concept – the translation of 2nd to

3rd generation test cases entails different criticalities if applied to web-based or mobile test cases, and can be used as a technique to mitigate different issues that are domain-specific.

7.2. Repair-based approaches

Many studies in the literature have focused on a repair-based approach, aiming at correcting locators or instructions in test cases that fail when the AUT changes. Imtiaz et al. highlight the main trends in the field of studying test scripts repairing automation, applied on the web domain [53]. On the other hand, few tools were specific to the GUI testing of mobile applications. Li et al. introduce ATOM, a tool for automated maintenance of test scripts for mobile applications [54]. To perform this task, ATOM uses two different models: an *Event Sequence Model* (ESM) and a *Delta ESM* (DESM), that respectively represent a possible event sequence and the possible changes done on the GUI transitioning from a version of the application to the next one.

CHATEM [3] extends ATOM to implement change-based testing. In practice, taking two different versions of the same application (e.g., two consecutive releases), the tool can extract the changes between the two GUIs and to generate maintenance actions for each change, combining them to create repair actions for the broken test scripts.

The described tools, compared with TOGGLE, are specifically tailored to solve the issue of (graphic) change-related fragility and need the extraction of a model of the user interface to enable the repair of broken test suites.

7.3. Computer vision-based approaches

Several studies have designed approaches based on computer vision to adapt test suites designed for a given SUT on different devices. Thereby, those studies aimed at reducing the costs for tackling the issue of fragmentation of visual test cases. Yu et al. described LIRAT [55], an image-driven tool that aims at recording and replay test scripts for the same application on different devices and platforms. The tool is based on image understanding techniques (namely, the SIFT feature extraction algorithm and KNN) to locate similar images on different renditions of the same GUI. Differently from the translation-based approach we propose, the tool does not take existing layout-based test cases as input, but instead relies on a single-step Script Recording phase performed by the tester/developer at the beginning of the process. Tuovenen et al. have described MAuto [56], a tool for the creation of cross-device visual test cases for Android apps. MAuto uses AKAZE features and is primarily tailored to reproduce user interaction with mobile games.

Behrang and Orso described AppTestMigrator [57], a tool that attempts to automatically transform a sequence of events and oracles designed for a specific app to other similar applications. AppTestMigrator leverages commonalities between user interfaces to automatically migrate existing tests written for an app to another similar app.

Cardenas et al. developed a tool named V2S [58] which generates replayable test scripts from video recordings of Android applications. The tool is primarily based on computer vision techniques.

8. Threats to Validity

Threats to Construct Validity: We have considered the success rate as a proxy for the evaluation of the precision of test cases, i.e., we expect that tests for working features must pass.

The results about the performance of the generated 3$^{\text{rd}}$ generation test scripts are influenced by the static sleep instructions added during the translation of 2$^{\text{nd}}$ generation scripts, which by converse need no explicit sleep instructions. The reports about the net time for interaction that are reported can just be used as an estimate of the lowest possible time for performing an interaction with the proposed Visual tools since a time interval for the rendering of the user interface, after the execution of commands, is not avoidable. In future enhancements of the tool, the sleep instructions should be made dynamic, utilizing GUI-state information to determine changes in the rendered screen before searching for visual locators. Dynamic sleep instructions are perceived to help the performance by mitigating unnecessary waiting time between consecutive interactions.

Threats to Conclusion Validity: To verify the presence of a statistically significant difference among different target tools, we applied standard statistical tests. The results are clear cut and consistent with the visual representations that report standard (95%) confidence intervals or complete distributions.

Researcher bias is another possible threat to the validity of this study since it involved a comparison in terms of different metrics of different 3$^{\text{rd}}$ generation testing tools. However, the authors have no reason to favour any particular approach, neither inclined to demonstrate any specific result.

Threats to External Validity: We recognize that the documented experimental design includes some bias as only interactions supported by the translator were used, i.e., only the Espresso commands belonging to the *OnView* family. The results of this evaluation are hence theoretically not generalizable to any Espresso test suite. However, TOGGLE's array of supported interactions include the most common ones used in Espresso, determined by an analysis of a set of 22,000 Espresso test files extracted from GitHub. Specifically, on the examined set, 97.33% of commands belonged to the *OnView* set, with just the remaining 2.67% belonging to the *OnData* set, not supported by the tool we developed. This finding indicates that our results would be applicable to the vast majority of test cases developed from open-source developers.

We also performed a statistical analysis, to ensure that the set of properties and actions we used is representative of what is widely used in available GitHub repositories. Hence, we applied the Chi-Square tests to verify the Null Hypotheses $H0_{va}$*: The View Actions in the developed test suites and the View Actions used in test cases mined from open-source repositories do not belong to the same* *distribution*, and $H0_{vi}$*: The View Identifiers in the developed test cases and the View Actions used in test suites mined from open-source repositories do not belong to the same distribution.* We could reject both null hypotheses ($p < 10^{-16}$), hence we can assume that the View Actions and Interactions we used belonged to the same distribution of those in tests mined from open-source repositories.

The approach we developed is based on the assumption that the state of the application is always reflected by the pictorial GUI shown to the user. This means that 2nd generation test cases containing assertion on a lower level of abstraction (i.e., internal properties of the widgets or values of the variables declared in the code of Activities) cannot be entirely translated to equivalent 3rd generation test cases, that cannot verify state changes that are not reflected by the appearance of the widgets of the graphical hierarchy. As well, Espresso test cases may contain direct interaction with methods declared in Activities without passing through widget interaction. These instructions do not have a visual testing counterpart. However, using these instructions in Espresso would result in developing unit tests of the SUT instead of pure GUI layout-based test cases, thereby having test artefacts that are by construct not eligible to be reproduced by visual test script drivers.

As of now, the findings of the experimental section apply to the considered 2nd and 3rd generation testing tools only, limiting the external validity of this work. However, it is possible to extend the syntaxes supported by TOGGLE by taking into consideration other testing tools, especially existing GUI Automation Frameworks for Android (e.g., Appium, or UIAutomator). Additionally, it is not assured whether the precision, performance, and fragility reduction values would be the same if measured with different typologies of applications with a very different graphical appearance compared to those that were considered (e.g., very graphically intensive projects such as games or video players). As well, the measured execution times proved to be strongly dependent on the type of interactions executed on the AUT, hence lowering the external validity of the results.

9. Conclusion and Future Work

In this work, we proposed the proof of concept of a novel approach for the creation of visual test cases in the mobile domain. The approach has been implementated in a tool called TOGGLE. The tool can translate layout-based 2nd generation test cases, written in Espresso, to visual 3rd generation test cases using the SikuliX and EyeAutomate syntax. Similar approaches have previously been evaluated in the field of web applications and DOM-based testing. However, to the best of our knowledge, this represents the first work in the literature about the translation-based generation of GUI test cases for mobile applications.

To investigate the feasibility of the approach and its capability in overcoming known limitations of visual testing for mobile apps, we have experimented with five test suites that we developed for as many popular Android open-source applications. The tool was able to generate working test cases with high precision and high success rate. It demonstrated that it is possible to reduce the testers' maintenance and development efforts by reusing existing layout-based test suites to create and maintain visual ones.

In addition to fixing some current limitations of the tool at its current state of development, the natural prosecution of this work will be an evaluation of the approach in a real industrial environment, to quantify its benefits in the creation and maintenance of real-world test suites.

As other future steps, we also identify the evaluation of an inverse translator, able to define layout-based test suites from existing visual ones. The backward translation would provide the added benefits of a possible creation of 2$^{\text{nd}}$ generation test scripts through reuse of existing 3$^{\text{rd}}$ generation counterparts, and the mitigation of layout-based fragilities (i.e., changed 2$^{\text{nd}}$ generation locators invalidating layout-based test cases) by re-translation from 3$^{\text{rd}}$ generation tests that are still valid. This feature would allow a significant reduction of the maintenance cost of layout-based test suites, for which the impact of fragilities is known to be relevant [37].

Also, we plan to provide companion translators, compatible with test scripts written with other layout-based testing tools, and to extend the approach to hybrid/web-based Android apps.

References

[1] I. Banerjee, B. Nguyen, V. Garousi, A. Memon, Graphical user interface (gui) testing: Systematic mapping and repository, Information and Software Technology 55 (10) (2013) 1679–1694.

[2] E. Alégroth, Z. Gao, R. Oliveira, 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.

[3] N. Chang, L. Wang, Y. Pei, S. K. Mondal, 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.

[4] L. Wei, Y. Liu, S.-C. Cheung, Taming android fragmentation: Characterizing and detecting compatibility issues for android apps, in: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, 2016, pp. 226–237.

[5] M. Kamran, J. Rashid, M. W. Nisar, Android fragmentation classification, causes, problems and solutions, International Journal of Computer Science and Information Security 14 (9) (2016) 992.

[6] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, D. Lo, Understanding the test automation culture of app developers, in: 2015 IEEE 8th

International Conference on Software Testing, Verification and Validation (ICST), 2015, pp. 1–10.

[7] M. Leotta, A. Stocco, F. Ricca, P. Tonella, Pesto: Automated migration of dom-based web tests towards the visual approach, Software Testing, Verification And Reliability 28 (4) (2018) e1665.

[8] R. Coppola, L. Ardito, M. Torchiano, M. Morisio, Mobile testing: New challenges and perceived difficulties from developers of the italian industry, IT PROFESSIONAL (To appear) 6.

[9] L. Ardito, R. Coppola, M. Torchiano, 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.

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

[11] M. Linares-Vásquez, K. Moran, 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.

[12] B. Sadeh, K. Ørbekk, M. M. Eide, N. C. Gjerde, T. A. Tønnesland, 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.

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

[14] S. Negara, N. Esfahani, 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.

[15] L. Gomez, I. Neamtiu, T. Azim, 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.

[16] Y. Hu, T. Azim, I. Neamtiu, Versatile yet lightweight record-and-replay for android, SIGPLAN Not. 50 (10) (2015) 349–366. doi:10.1145/2858965.2814320.
URL `https://doi.org/10.1145/2858965.2814320`

[17] M. Halpern, Y. Zhu, R. Peri, V. J. Reddi, Mosaic: cross-platform user-interaction record and replay for the fragmented android ecosystem, in: Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on, IEEE, 2015, pp. 215–224.

[18] M. Fazzini, E. N. d. A. Freitas, S. R. Choudhary, 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.

[19] K. Moran, R. Bonett, C. Bernal-Cárdenas, B. Otten, D. Park, D. Poshyvanyk, On-device bug reporting for android applications, in: Mobile Software Engineering and Systems (MOBILESoft), 2017 IEEE/ACM 4th International Conference on, IEEE, 2017, pp. 215–216.

[20] K. Mao, M. Harman, 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.

[21] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, 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.

[22] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, 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.

[23] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, A. M. Memon, Mobiguitar: Automated model-based testing of mobile apps, IEEE software 32 (5) (2015) 53–59.

[24] T. Yeh, T.-H. Chang, 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.

[25] E. Alegroth, M. Nass, 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.

[26] E. Alégroth, A. Karlsson, A. Radway, Continuous integration and visual gui testing: Benefits and drawbacks in industrial practice, in: Software Testing, Verification and Validation (ICST), 2018 IEEE 11th International Conference on, IEEE, 2018, pp. 172–181.

[27] E. Alégroth, R. Feldt, P. Kolström, Maintenance of automated test suites in industry: An empirical study on visual gui testing, Information and Software Technology 73 (2016) 66–80.

[28] E. Alégroth, R. Feldt, On the long-term use of visual gui testing in industrial practice: a case study, Empirical Software Engineering 22 (6) (2017) 2937–2971.

[29] F. Dobslaw, R. Feldt, D. Michaelsson, P. Haar, F. G. Neto, R. Torkar, Estimating return on investment for gui test automation tools, arXiv preprint arXiv:1907.03475 (2019).

[30] Y.-D. Lin, J. F. Rojas, E. T.-H. Chu, Y.-C. Lai, On the accuracy, efficiency, and reusability of automated test oracles for android devices, IEEE Transactions on Software Engineering 40 (10) (2014) 957–970.

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

[32] V. Garousi, M. Felderer, Developing, verifying, and maintaining high-quality automated test scripts, IEEE Software 33 (3) (2016) 68–75.

[33] A. M. Memon, Automatically repairing event sequence-based gui test suites for regression testing, ACM Transactions on Software Engineering and Methodology (TOSEM) 18 (2) (2008) 4.

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

[35] D. Han, C. Zhang, X. Fan, A. Hindle, K. Wong, E. Stroulia, Understanding android fragmentation with topic analysis of vendor-specific bugs, in: Reverse Engineering (WCRE), 2012 19th Working Conference on, IEEE, 2012, pp. 83–92.

[36] J.-H. Park, Y. B. Park, H. K. Ham, Fragmentation problem in android, in: 2013 International Conference on Information Science and Applications (ICISA), IEEE, 2013, pp. 1–2.

[37] R. Coppola, M. Morisio, M. Torchiano, Mobile gui testing fragility: A study on open-source android applications, IEEE Transactions on Reliability (2018).

[38] H. Zheng, D. Li, B. Liang, X. Zeng, W. Zheng, Y. Deng, W. Lam, W. Yang, T. Xie, Automated test input generation for android: towards getting there in an industrial case, in: Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), 2017 IEEE/ACM 39th International Conference on, IEEE, 2017, pp. 253–262.

[39] Y. Liu, C. Xu, S.-C. Cheung, Characterizing and detecting performance bugs for smartphone applications, in: Proceedings of the 36th international conference on software engineering, 2014, pp. 1013–1024.

[40] C. Wilke, C. Piechnick, S. Richly, G. Püschel, S. Götz, U. Aßmann, Comparing mobile applications' energy consumption, in: Proceedings of the 28th Annual ACM Symposium on Applied Computing, 2013, pp. 1177–1179.

[41] A. Hovsepyan, R. Scandariato, W. Joosen, J. Walden, Software vulnerability prediction using text analysis techniques, in: Proceedings of the 4th international workshop on Security measurements and metrics, 2012, pp. 7–10.

[42] N. Mathur, S. A. Karre, Y. R. Reddy, Usability evaluation framework for mobile apps using code analysis, in: Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018, ACM, 2018, pp. 187–192.

[43] R. Feng, G. Meng, X. Xie, T. Su, Y. Liu, S.-W. Lin, Learning performance optimization from code changes for android apps, in: 2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), IEEE, 2019, pp. 285–290.

[44] L. Ardito, R. Coppola, M. Morisio, 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.

[45] K. Srisopha, R. Alfayez, Software quality through the eyes of the end-user and static analysis tools: a study on android oss applications, in: Proceedings of the 1st International Workshop on Software Qualities and Their Dependencies, ACM, 2018, pp. 1–4.

[46] J. Ferreira, A. C. Paiva, Android testing crawler, in: International Conference on the Quality of Information and Communications Technology, Springer, 2019, pp. 313–326.

[47] R Core Team, R: A Language and Environment for Statistical Computing, R Foundation for Statistical Computing, Vienna, Austria (2018).
URL `https://www.R-project.org/`

[48] E. Borjesson, R. Feldt, Automated system testing using visual gui testing tools: A comparative study in industry, in: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, IEEE, 2012, pp. 350–359.

[49] C. Sacramento, A. C. Paiva, Web application model generation through reverse engineering and ui pattern inferring, in: 2014 9th International Conference on the Quality of Information and Communications Technology, IEEE, 2014, pp. 105–115.

[50] W. Yang, M. R. Prasad, T. Xie, A grey-box approach for automated gui-model generation of mobile applications, in: International Conference on Fundamental Approaches to Software Engineering, Springer, 2013, pp. 250–265.

[51] M. Leotta, A. Stocco, F. Ricca, P. Tonella, Using multi-locators to increase the robustness of web test cases, in: 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), IEEE, 2015, pp. 1–10.

[52] A. Stocco, M. Leotta, F. Ricca, P. Tonella, Pesto: A tool for migrating dom-based to visual web tests, in: 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation, IEEE, 2014, pp. 65–70.

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

[54] X. Li, N. Chang, Y. Wang, H. Huang, Y. Pei, L. Wang, 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.

[55] S. Yu, C. Fang, Y. Feng, W. Zhao, Z. Chen, Lirat: Layout and image recognition driving automated mobile testing of cross-platform, in: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2019, pp. 1066–1069.

[56] J. Tuovenen, M. Oussalah, P. Kostakos, Mauto: Automatic mobile game testing tool using image-matching based approach, The Computer Games Journal 8 (3-4) (2019) 215–239.

[57] F. Behrang, A. Orso, Test migration between mobile apps with similar functionality, in: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2019, pp. 54–65.

[58] C. Bernal-Cárdenas, N. Cooper, K. Moran, O. Chaparro, A. Marcus, D. Poshyvanyk, Translating video recordings of mobile app usages into replayable scenarios, in: Proc. of 42nd Int. Conf. on Software Engineering, 2020, p. 13. doi:10.1145/ 3377811.3380328.

1680 ## Appendix A. Translation to 3rd-generation specific syntax

Table A.16: TOGGLE - 3rd generation test script creator: Translation from Tool-agnostic instructions to Tool-specific commands

Logged interaction	**EyeAutomate commands**	**Sikuli commands**
clearText	i. Click *img* ii. Type [BACKSPACE] (*arg1* times)	i. click(*img*) ii. type(Key.BACKSPACE) (*arg1* times)
click	i. Click *img*	i. click(*img*)
closesoftkeyboard	i. Type [CTRL_PRESS] ii. Sleep 10 iii. Type [BACKSPACE] iv. Sleep 10 v. Type [CTRL_RELEASE]	i. keyDown(Key.CTRL) ii. sleep(0.01) iii. type(Key.BACKSPACE) iv. sleep(0.01) v. keyUp(Key.CTRL)
doubleclick	i. MouseDoubleClick *img* i. Click *img* ii. Type *arg1*	i. hover(*img*) ii. mouseDown(Button.LEFT) iii. sleep(0.001) iv. mouseUp(Button.LEFT) v. sleep(0.001) vi. mouseDown(Button.LEFT) vii. sleep(0.001) viii. mouseUp(Button.LEFT)
longclick	i. Move *img* ii. MouseLeftPress iii. Sleep 500 iv. MouseLeftRelease	i. hover(*img*) ii. mouseDown(Button.LEFT) iii. sleep(0.5) iv. mouseUp(Button.LEFT)
typetext	i. Click *img* ii. Type *arg1*	i. click(*img*) ii. type(*arg2*)
openactionbarmenu	i. Type [CTRL_PRESS] ii. Sleep 10 iii. Type m iv. Sleep 10 v. Type [CTRL_RELEASE]	i. keyDown(Key.CTRL) ii. sleep(0.01) iii. type(m) iv. sleep(0.01) v. keyUp(Key.CTRL)
pressback	i. Type [CTRL_PRESS] ii. Sleep 10 iii. Type [BACKSPACE] iv. Sleep 10 v. Type [CTRL_RELEASE]	i. keyDown(Key.CTRL) ii. sleep(0.01) iii. type(Key.BACKSPACE) iv. sleep(0.01) v. keyUp(Key.CTRL)
presskey	i. Type *arg1*	i. type(*arg1*)
pressmenukey	i. Type [CTRL_PRESS] ii. Sleep 10 iii. Type h iv. Sleep 10 v. Type [CTRL_RELEASE]	i. keyDown(Key.CTRL) ii. sleep(0.01) iii. type(h) iv. sleep(0.01) v. keyUp(Key.CTRL)
replacetext	i. Click *img* ii. Type [BACKSPACE] (*arg1* times) iii. Type *arg2*	i. click(*img*) ii. type(Key.BACKSPACE) (*arg1* times) iii. type(*arg2*)
swipedown[5]	i. Move *img* ii. Sleep 10 iii. MouseLeftPress iv. MoveRelative "0" "250" v. MouseLeftRelease	i. r = find(*img*) ii. start = r.getCenter() iii. stepY = 250 iv. run = start v. mouseMove(start); wait(0.2) vi. mouseDown(Button.LEFT); wait (0.2) vii. run = run.below(stepY) viii. mouseMove(run) ix. mouseUp() xi. wait(0.2)

[5]For better conciseness, we only report as an example the Swipe Down instruction. The tool also translates swipes with Left, Right and Up directions, with adaptations in the relative movements performed by the mouse.