

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

Original

Translation from Layout-based to Visual Android Test Scripts: an Empirical Evaluation / Coppola, R., Ardito, L., Torchiano, M., Alégroth, E.. - In: THE JOURNAL OF SYSTEMS AND SOFTWARE. - ISSN 0164-1212. - ELETTRONICO. - 171:(2021), pp. 1-26. [10.1016/j.jss.2020.110845]

Availability:

This version is available at: 11583/2848261 since: 2021-06-22T12:02:00Z

Publisher:

Elsevier

Published

DOI:10.1016/j.jss.2020.110845

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

Elsevier postprint/Author's Accepted Manuscript

© 2021. This manuscript version is made available under the CC-BY-NC-ND 4.0 license
<http://creativecommons.org/licenses/by-nc-nd/4.0/>. The final authenticated version is available online at:
<http://dx.doi.org/10.1016/j.jss.2020.110845>

(Article begins on next page)

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

2 The Android operating system has recently reached its ninth release and has
3 been confirmed as the platform of choice for nearly 90% of mobile users as of the
4 first half of 2019. Modern Android applications (henceforth referred to as apps)
5 are complex, generally on par with desktop software with interactive graphical
6 user interfaces (GUI) and large-scale server back-ends. Similar to desktop soft-
7 ware, apps are also developed using modern development processes in quick and

8 short delivery cycles. Short deliveries that make quick, and thorough, verifica-
 9 tion and validation phases crucial in both open-source and industrial settings.
 10 Android apps are also GUI-intensive, putting emphasis on testing their visual
 11 correctness in addition to their functional behaviour.

12 During the last ten years, many end-to-end (from now on referred to as *E2E*)
 13 testing tools have been proposed for Android app testing. E2E tests are defined
 14 as repeatable test scripts that automate the interaction with the application
 15 as a whole, without isolating its components (i.e. a black-box approach), to
 16 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()));

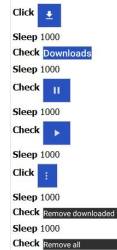
    onView(withId(R.id.pause_downloads)).check(matches(isDisplayed()));
    onView(withId(R.id.resume_downloads)).check(matches(isDisplayed()));

    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



(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

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

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

30 Because of the limitations of 2nd generation tools, 3rd generation testing tools
 31 have been proposed that use image recognition technology to test the apps' vi-
 32 sual appearance or, more commonly, their behaviour through the pictorial user
 33 interface. These script-based test scenarios therefore include screen captures
 34 (see fig. 1b) that are used as locators to identify widgets. The screen cap-
 35 tures are also used as oracles that compare the current appearance of the app
 36 to the visually expected result after the interactions are performed. Because
 37 3rd generation tools rely on image recognition, they are, in contrast to 2nd ge-
 38 neration tools, agnostic to platform/system/programming language, requiring
 39 only access to the pictorial GUI of the SUT to run tests. However, compared

40 to 2nd generation, because of the computationally heavy and imprecise image
41 recognition algorithms, tools of this approach generally have lower test execution
42 time performance and lower robustness to graphical change.

43 2nd and 3rd generation testing tools currently coexist in practice of the testing
44 community, although 2nd generation tools are more common than 3rd generation
45 ones. We also stress that the adopted categorization of the generations of GUI
46 testing approaches is strictly chronological and do not reflect that later gener-
47 ations should be more effective/efficient. For instance, image-recognition based
48 tools (3rd generation) are not considered more effective/efficient or a replace-
49 ment for Layout-based (2nd generation) tools for GUI-based testing. Research
50 into 2nd and 3rd generation tools has instead shown that they have complemen-
51 tary benefits and characteristics [2].

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

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

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

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

77 In this paper, we investigate the creation of 3rd generation Android app
78 test scripts by translating them from 2nd generation test scripts that are used
79 as templates. We base this approach on the premise that 2nd and 3rd gener-
80 ation tools share several commonalities in terms of the test structure, such
81 as step-wise test sequences of interactions/assertions of widgets, similar timing
82 constraints and similar test purpose for functional tests. The main objective
83 of this work is therefore to improve the value of existing 2nd generation test
84 scripts by providing practitioners with a cost-efficient extension of their existing
85 2nd generation testing capabilities to 3rd generation testing as well.

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

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

- 93 • A test creation approach built on the translation of 2nd to 3rd generati-
94 on GUI test cases for Android apps. GUI test translation has previously
95 been demonstrated for Web-based applications [7] but, to the best of our
96 knowledge, not for Android apps except for our previous work that served
97 as the basis of the work presented here [8];
- 98 • The general architecture and implementation details of a tool that demon-
99 strates the approach, TOGGLE. The implementation details are comple-
100 mented with an extended proof of concept study of the tool based on
101 previous research [9];
- 102 • Results from evaluation of the success rate of the implemented approach in
103 the translation of 2nd generation test scripts to 3rd generation scripts, the
104 translated tests' execution success rate on Android apps and the ability
105 of the approach to mitigate graphic and fragmentation-induced fragility.
- 106 • As a side-effect of this work, the study shows how existing 3rd generati-
107 on testing tools can be applied to Android applications, which provides a
108 contribution to 3rd generation software testing research [10].

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

119 2. Background

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

123 *2.1. Layout-based (2nd generation) testing of Android apps*

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

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

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

146 Other open-source and widely-adopted alternatives in the literature are Robolec-
147 tric [12] and Robotium [13].

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

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

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

163 *2.2. Visual (3rd generation) testing tools*

164 Third generation testing tools can automate any graphical user interface
165 using screen captures of the individual widgets, which are used both as locators
166 and oracles to verify the state of the AUT after several interactions. These

167 tools are mostly agnostic to the implementation of the AUT, and they can,
168 therefore, be used to automate any kind of application provided with a GUI –
169 given that it is emulated on a desktop pc where the visual recognition engine
170 can be run. Some examples of general-purpose 3rd generation GUI testing tools
171 are SikuliX [24], EyeAutomate (evolution of JAutomate [25]), or AppliTools.

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

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

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

187 2.3. Challenges in Android automated testing

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

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

206 A GUI test case can be defined *fragile* if it requires intervention when the
207 application evolves (i.e., between two consecutive releases) due to any modifi-
208 cation applied to the SUT [32][33]. As stated, mobile test cases are also heav-
209 ily subject to fragilities since frequent changes are applied to the GUI during
210 the app’s lifespan and test cases defined with 2nd or 3rd generation automation

211 frameworks are strictly tied to it. Many different causes can concur with the
212 fragility in GUI test cases: in our previous works, we defined a taxonomy of 30+
213 types of actions on the AUT that may trigger test fragilities [34]. At a higher
214 level, we note that it is possible to distinguish between 2nd generation-related
215 fragilities when changes are applied to the widget definition thus causing failures
216 in 2nd generation test cases, and 3rd generation-related fragilities, when visual
217 modifications are performed on the pictorial GUI, and hence visual locator may
218 not be found.

219 The *Fragmentation* issue includes two different concepts [35]. First, *Hardware-*
220 *based* fragmentation is related to the fact that any Android app must be run
221 on different devices, with varying hardware specifications. Hardware fragmen-
222 tation has a major impact on 3rd generation (Visual) testing since also screen
223 sizes, and pixel densities change significantly between one device and another.
224 A valid locator or oracle for one device may therefore be unusable on a device
225 where the same image is rendered at a different pixel density. Additionally, An-
226 droid allows the developers to define different layout files for the same activities
227 that are inflated based on the specific screen size or orientation of the device
228 where the application is run. This type of device-related variability may impact
229 2nd and 3rd generation test cases that can be invalidated because the
230 widgets with scripted interactions are rendered in different ways or substituted
231 with other components. Hardware fragmentation, thus, has high costs on the
232 practice of testing, because test cases should be re-recorded, or at least verified,
233 on each of the devices with which the AUT must be compatible.

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

240 3. Motivation

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

- 250 • More complex native interactions (e.g. hand gestures) that do not natu-
251 rally translate to the mouse/keyboard inputs offered by most 3rd genera-
252 tion tools.

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

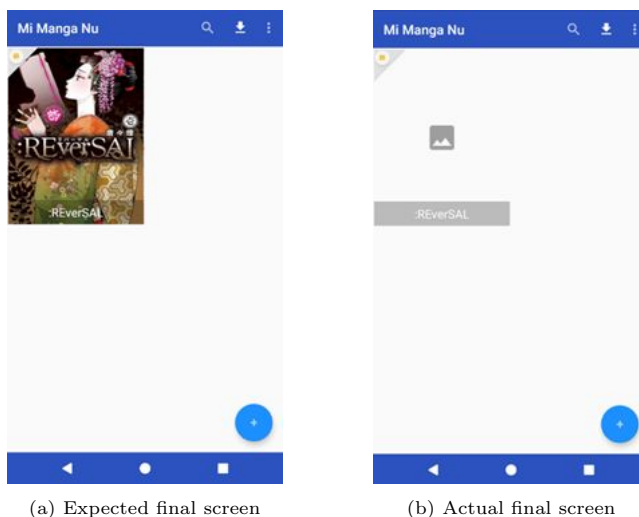


Figure 2: Example of 3rd generation true positive and 2nd generation false negative

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

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

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

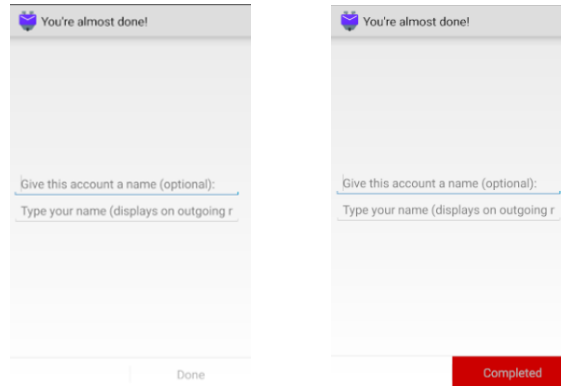
276 • **A fault is present and only 2nd generation fails whilst 3rd ge-**
277 **neration passes:** In this case, the 2nd generation reports a true positive
278 whilst the 3rd generation reports a false negative. Faults of this type can be
279 related to specific widget properties, e.g. change of ID numbers,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

282 • **A fault is present but the 2nd generation reports a pass whilst**
283 **3rd generation fails:** In this case, the 3rd generation test case reports a
284 true positive whilst the 2nd generation test case reports a false negative.
285 Faults of this type generally relate to the visual appearance of the app
286 and are not verifiable by 2nd generation test assertions. Figure 2 presents
287 an example where sub-figure “a” reports the expected output whilst sub-
288 figure “b” shows the actual output. The cause of the test result discrep-
289 ancy could, for instance, be that the graphics library failed to load the
290 image to the container. The 2nd generation test is only able to verify that
291 the container is rendered, but not its visual content, and therefore passes
292 incorrectly.

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

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

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



(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

- 311 • **Reduced impact of graphic fragilities:** Similarly to fragmentation,
312 translation-based creation can help in solving fragilities caused by visual
313 changes to the GUI over time through continuous re-translation of
314 3rd generation tests from 2nd generation test cases. Whilst this limits the
315 regression-testing capability for the version of the app on which the trans-
316 lation occurred, the benefits of automatic visual testing can still be reaped.

317 Figure 3 shows an example of fragility where the text and background
318 colour of a button has been changed. The 2nd generation test is still valid
319 because it disregards the visual appearance, but a previously translated
320 3rd generation test would fail, reporting a false positive. As such, in this
321 case, re-translation would be required for a new test that could, given
322 that this change remains in the next version of the app, be used for visual
323 regression testing.

324 4. TOGGLE

325 We implemented the translation-based approach in a tool, TOGGLE (Trans-
326 lation Of Generations of GUI testing at Low Effort). The core idea behind the
327 proposed translator is to use the information provided by 2nd generation test
328 scripts to create 3rd generation scripts. A first theoretical proof-of-concept of
329 the translation approach (including the design for a backward translator, from
330 3rd to 2nd generation test scripts) has been presented in our previous work [9].
331 There we provided a high-level description of the building blocks of the architec-
332 ture, and we conceptually validated the approach by modifying and translating
333 manually 2nd generation test scripts. With the present work, we detail the ac-
334 tual implementation of the framework, and we evaluate it with real test cases
335 developed for Android apps.

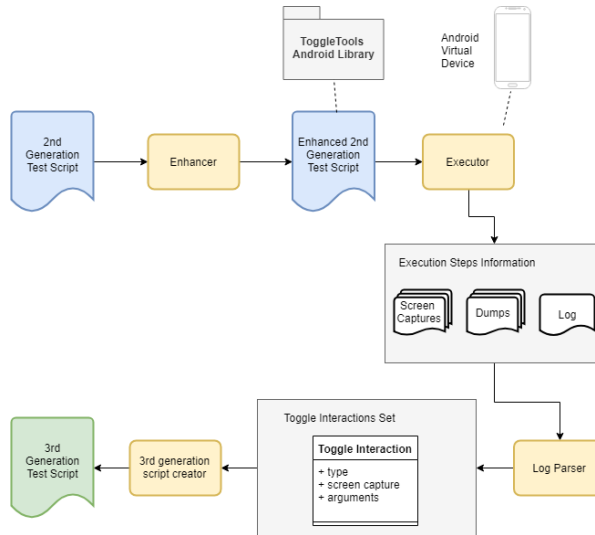


Figure 4: Architecture of TOGGLE for translation from 2nd to 3rd generation test scripts

336 The translation procedure is split into two parts. First, the test scenario
 337 – a series of GUI interactions and checks – is obtained through the execution
 338 and examination of a 2nd generation tests script. Second, the GUI interactions
 339 are identified, abstracted, and finally translated into the syntax of the target
 340 3rd generation tool. Theoretically, the approach can be applied to any 2nd generation
 341 syntax, given that a module capable of parsing the specific syntax of the
 342 tool is provided. Similarly, the output 3rd generation script can be created using
 343 the syntax of different test drivers, given that a module for the creation of the
 344 scripts is developed. In our implementation, we selected Espresso as the origin
 345 2nd generation testing tool, because it emerged from the literature as one of the
 346 most adopted tools among open-source developers [37][6]. As target 3rd generation
 347 tools, we provided translation mechanisms to both EyeAutomate and
 348 SikuliX, since they are the most cited in empirical studies about visual testing.

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

351 The high-level architecture contains four main modules:

- 352 • **Enhancer:** it parses a 2nd generation test script, to inject function calls
 353 from the TOGGLE library into the code. This is required to extract
 354 screen captures and XML files containing the dump of the current screen
 355 hierarchy (from now on simply referred to as *dumps*);
- 356 • **Executor:** it executes the enhanced 2nd generation script on a real or
 357 emulated Android Virtual Device, checking the outcome of the test whilst
 358 saving screen captures and screen hierarchy dumps on the device memory,
 359 while logging the trace of the performed interactions;

- 360 • **Log Parser:** it parses the log saved from the executor, reconstructing the
361 properties of each interaction and finding the exact visual locators to use
362 in the 3rd generation test cases;
- 363 • **Third generation script creator:** it translates the intermediate and
364 tool-agnostic sequence of interactions to the desired 3rd generation syntax.

365 The individual modules are detailed in the following subsections.

366 4.1. Enhancer

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

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

376 The inspection of 2nd generation test cases was performed using the Java-
377 Parser library¹, identifying method calls of 2nd generation interactions. For each
378 identified interaction, the following method calls from the TOGGLE library are
379 added:

- 380 • *TakeScreenCapture:* The method uses the UI Automator framework [38]
381 to take a capture of the current screen of the application. The full-screen
382 capture is saved, as a Bitmap file, in the emulated external storage of the
383 AVD. The screen capture is named after the test case name, followed by
384 a progressive identifier number.
- 385 • *DumpScreen:* The method uses the UI Automator framework to extract
386 the dump of the current screen hierarchy. The dump is an XML file,
387 which reports all the layout properties of the widgets that are shown on
388 the screen at a given time. The dump is saved in the emulated external
389 storage of the AVD. Similar to the corresponding screen capture, it is
390 named after the test case name, followed by a progressive id number.
- 391 • *LogInteraction:* The method uses the Android built-in LogCat tool, to log
392 information about the interaction that has been performed. The logged
393 line contains the following parameters: (i) *search_type*, i.e. the type of
394 widget property used as a locator (e.g., "id", "text", "content-desc");
395 (ii) *search_keyword*, i.e. the specific value of the locator (e.g., the id
396 "search.button"); (iii) *interaction_type*, i.e. the type of interaction per-
397 formed on the widget (e.g., "click", "type-text"); (iv) *interaction_params*,

¹<https://github.com/javaparser/javaparser>

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

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

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

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

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

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

436 The enhanced 2nd generation test case also includes a sleep instruction be-
437 tween the interactions. These sleep instructions are not added to the created
438 3rd generation test cases, they are only present in the enhanced test cases to
439 allow the system to have the time to obtain the screen captures and dumps.
440 Since this sleep instruction only impacts the translation phase of the script and
441 not the execution of the visual test scripts, we have adopted a fixed sleep time of
442 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

443 of screen captures on the storage of the emulated devices.

444 *4.2. Executor*

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

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

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

467 *4.3. Log parser*

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

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

- 473 1. It reads an interaction from the log, retrieving its parameters;
- 474 2. Using the progressive number of the interaction inside the test case, it
475 retrieves the screen hierarchy dump which was created in the external
476 storage at runtime;
- 477 3. Searches in the dump files for the interaction parameters (i.e., searches
478 for a widget with the value of *search_type* equal to *search_keyword*, and
479 extract the boundaries of the interacted widget). This step allows more
480 precise captures of the location where the widget has been rendered at
481 runtime. Thus, eliminating problem factors such as what device the app
482 was launched on, the apps orientation, the position of the element in a
483 list, etc. Hence, information that cannot be retrieved from static analysis
484 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,

```

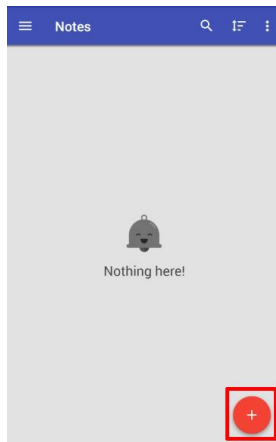
(a) Log excerpt

```

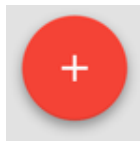
<node bounds="[0,1269][1080,1794]" visible-to-
user="true" selected="false" password="false" long-
clickable="false" scrollable="false" focused="false"
focusable="false" enabled="true" clickable="false"
checked="false" checkable="false" content-desc=""
package="it.feio.android.omninetes.foss"
class="android.view.ViewGroup" resource-
id="it.feio.android.omninetes.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.omninetes.foss"
class="android.view.ViewGroup" resource-
id="it.feio.android.omninetes.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.omninetes.foss"
class="android.widget.ImageButton" resource-
id="it.feio.android.omninetes.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

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

490 We report an example of the operations performed by the Log Parser in
 491 figure 6: starting from the first instruction found in the log (fig. 6a), the Log
 492 Parser identifies the exact widget inside the hierarchy dump (highlighted in fig.
 493 6b), then uses the full screen capture of the screen to cut the exact visual locator
 494 for the widget (highlighted in figs. 6c and 6d). This visual locator, paired with
 495 the interaction info, will be the output used to create 3rd generation scripts.

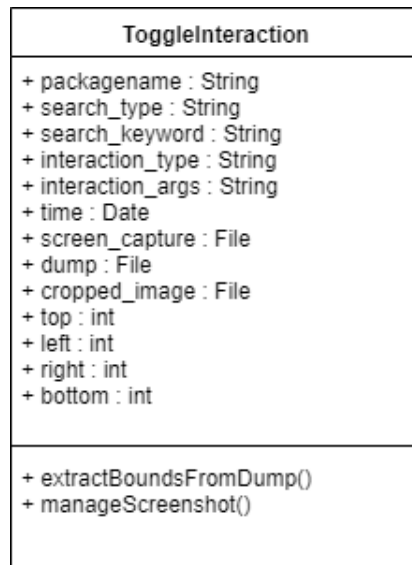


Figure 7: The TOGGLEInteraction Class

496 The information characterizing each set of widget properties is stored in-
 497 side a TOGGLEInteraction object. This format is a completely tool-agnostic
 498 representation of each interaction with the device. The format of the TOG-
 499 GLEInteraction object is shown in figure 7; the fields contain the following
 500 information: *packagename* is the name of the tested .apk, concatenated to the
 501 name of the test file, to differentiate between different test sessions; *search.type*,
 502 *search.keyword*, *interaction.type*, *interaction.args* are the field retrieved from
 503 the log line related to the interaction; *time* is the timestamp at the moment of
 504 the execution, and serves as a unique id for the interaction; *screen_capture* and
 505 *dump* are pointers to the files in external storage obtained during the execution;
 506 *cropped_image* is the visual locator for the interacted widget; *top*, *left*, *right*,
 507 *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

508 4.4. Third generation Script Creator

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

512 In general, a 1-to-1 mapping between 2nd generation interactions to 3rd ge-
 513 neration ones is not possible since 2nd generation interactions often act directly
 514 on the recognized views (e.g., insert a string directly inside a TextView without
 515 putting it in focus or access an item in a list which is not expanded). The
 516 development of the Script Creator module hence entails an analysis of what
 517 type of commands can be executed against the GUI of an Android app, to
 518 find the proper way of translating them into the commands featured by the
 519 3rd generation test drivers.

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

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

534 Since the transitions in the GUI may be not immediate, depending on the
 535 app characteristics, animations, and possible race conditions with other apps
 536 running on the emulated devices, we leverage commands of the target script
 537 syntaxes to dynamically wait for the appearance of the desired widgets. These
 538 commands wait for an amount of time, that can be fixed by the programmer
 539 before the test ends up in a failure. We have set this timeout to 30 seconds, a
 540 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

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

549 Another important design decision made for the Translator module was
 550 about where to insert the assertions in the created 3rd generation test scripts.
 551 2nd generation assertions can verify varying aspects of the widgets, e.g., their
 552 textual content or parameters like their visibility on screen, whereas 3rd generation
 553 tools can only verify the visual appearance of widgets. Starting from
 554 the assumption that the Enhanced test is executed on a stable version of the
 555 application, we resorted to capturing visual oracles for every assertion found in
 556 2nd generation code. Additionally, we added a final check of the whole screen
 557 at the end of each translated test script. This allows us to verify that the final
 558 appearance of the application, after the execution of all the test steps. A final
 559 full check is crucial to ensure that all the interactions of the test script were
 560 replayed as expected, because errors of the image recognition driver may lead
 561 3rd generation tools to perform intermediate operations on wrong elements of
 562 the GUI (because of similarities with the locators used in the script) without
 563 signalling any failure. Since the EyeAutomate library suffered from false posi-
 564 tives at the final full check, because of too many details in the images to locate,
 565 we added the possibility to tune the EyeAutomate recognition algorithm by
 566 changing the *Confirmation Threshold* parameter, which sets up the minimum
 567 similarity between the visual oracle and the rendered final screen to return a
 568 positive full check.

569 The output of the Script Creator is a visual test script, which can be run
 570 immediately against the app after its launch on an AVD to verify its appear-
 571 ance. Alternatively, the test script is added to an existing test suite for future
 572 regression testing. Hence, in addition to testing the system according to the
 573 same sequences as the 2nd generation test scripts, the visual scripts also verify
 574 the AUT's appearance.

575 At its current stage of development, TOGGLE supports translation to Eye-
 576 Automate and SikuliX scripts. The translated scripts have native formats for
 577 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

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

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

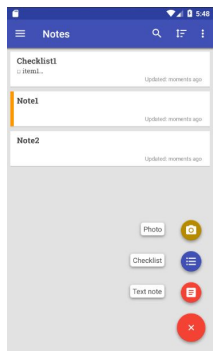
597 Table 4 summarizes the possible translations for 2nd generation test cases
598 that are currently supported by the 3rd generation script creator, along with the
599 acronyms that are used in the continuation of the paper. In the remainder of
600 the paper, we will indicate with *E* the original Espresso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

601 5. Evaluation

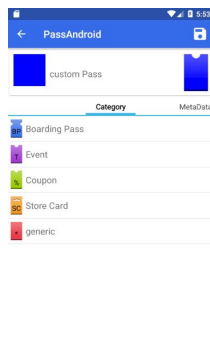
602 This section describes the experimental evaluation conducted on TOGGLE,
603 the adopted procedure and its results.

604 5.1. Experimental Subjects

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



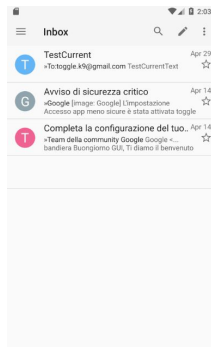
(a) Omni-Notes



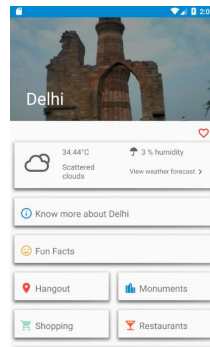
(b) PassAndroid



(c) MiMangaNu



(d) K9-Mail



(e) TravelMate

Figure 8: Screen captures of considered applications

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

615 The following criteria guided the apps selection:

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

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

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

- 631 • **K9-Mail:** a popular e-mail client, which has a long release history on the
632 GitHub platform. The application has been used by several experimental
633 studies in the field of mobile development and testing [39][40][41].
- 634 • **MiMangaNu:** an application for reading and organizing comics from
635 online repositories. It served as the example of an app with possible long-
636 running operations (the download operations of the comics) to see how
637 they were handled with the insertion of static sleep instructions. The app
638 is not available on the PlayStore. It has been used as an experimental
639 subject in related literature [42][43].
- 640 • **OmniNotes:** an application for managing text notes and checklists, with
641 possible multimedia attachments. The app is also available on F-Droid
642 and the PlayStore. We used this application as an experimental object
643 in one of our previous studies for the comparison of Second-generation to
644 Visual-based approaches [44], as well as in many other studies not limited
645 to the field of GUI testing [45][46].
- 646 • **PassAndroid:** an application for storing and managing different types of
647 tickets through QR codes. The app has a long release history on GitHub.
648 It is released on F-Droid and is also available for free on the PlayStore,
649 where it has more than a million downloads. We used release 2.5.0 because
650 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

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

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

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

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

669 We report in Table 6 the number of locators used in each test suite. In almost
670 half of the cases, the widgets had unique ids that could be used as locators. The
671 second choice as a locator, in terms of frequency of occurrence, was the textual
672 content of the widgets. Textual locators are however not as robust as id locators.
673 They are typically more prone to change during the evolution of the app, and
674 it is not possible to ensure their uniqueness on the screen. When the widgets
675 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

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

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

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

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

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

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

710 *5.2. Research Questions and Procedure*

711 The experimental evaluation aimed to answer the following research ques-
712 tions:

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

716 To answer RQ1, for each test case, we computed the Translation Time met-
717 ric, that we define as:

$$T_{tot} = T_{en} + T_{ex} + T_{sc} \quad (1)$$

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

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

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

$$P = \frac{I_{tr}}{N} \quad (2)$$

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

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

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

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

$$SR = E_s/E_t, \quad (3)$$

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

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

- 748 • **Passing**: when all the executions end with a success (i.e., $SR = 1$);
- 749 • **Flaky**: when some executions, but not all, end with a failure (i.e., $0 <$
 750 $SR < 1$);
- 751 • **Failing**: when all executions end with a failure (i.e., $SR = 0$).

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

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

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

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

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

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

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

780 While the goal is to detect differences among the tools, we include on the re-
 781 gression equation also the different apps to avoid the result being confounded
 782 by differences among them.

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

787 • **RQ4 - Visual Scripts Performance:** What is the performance of work-
 788 ing visual scripts in terms of average execution time?

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

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

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

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

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

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

818 The net time T_n can be deemed a more accurate estimate of the time em-
819 ployed by the studied algorithms for performing atomic Android commands on
820 the emulated AVD.

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

- 826 • **RQ5 - Robustness to Device Fragmentation Fragility:** What is
827 the advantage in terms of reduced fragility to device fragmentation when
828 generating 3rd generation test scripts by translation?

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

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

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

- 850 • **RQ6 - Robustness to Graphic Fragility:** What is the advantage in
851 terms of the reduced fragility to pure graphic changes when generating
852 3rd generation test scripts by translation?

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

856 For each application, we selected 15 distinct widgets to modify; we ap-
857 plied different kind of graphic changes. To select the modifications to apply,
858 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

859 scripts, that was defined in [34] by three of the authors. Within that taxon-
860 omy, only three categories of modifications can have an impact on the execution
861 of Visual test scripts: changes in the layout, changes in the text contained by
862 the widgets, pure graphic changes in the widget. In table 8 we report the sub-
863 categories of changes that we inferred by analyzing all the types of modifications
864 that can be performed in the layout information of any widget, and the number
865 of modifications applied to the five AUTs. Note that this may be higher than
866 15 since, in some cases, multiple variations were applied on a single widget.

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

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

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

880 5.3. Experimental setup

881 All the test cases have been run on a desktop PC with an Intel i7-8550U
882 at 1.80GHz clock, with 16GB RAM, and Windows 10 Operating System. The
883 development of the test suites and the execution of Espresso test cases were
884 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

885 emulated Nexus 5X API 25 (Android 7.11) with enabled device frame and key-
 886 board inputs. Animations were disabled on the AVD.

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

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

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

900 5.4. Experimental Results

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

905 In compliance with open science principles, we make available a replication
 906 package in the form of a code capsule².

907 5.4.1. RQ1 - Tool Performance

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

²Code 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

Application	T_{en}		T_{ex}		T_{sc}		T_t	
	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

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

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

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

928 **Answer to RQ1:** The translation-based approach, as implemented in Tog-
 gle, 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.

929 *5.4.2. RQ2 - Translation Precision*

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

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

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

Table 11: Command translation rate results

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

942 the added 2nd generation instructions required for the translation were not com-
 943 patible with the type of dialog boxes that were used in the traversed screens.

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

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

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

959

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.

960

961 5.4.3. RQ3 - Visual Scripts Success Rate

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

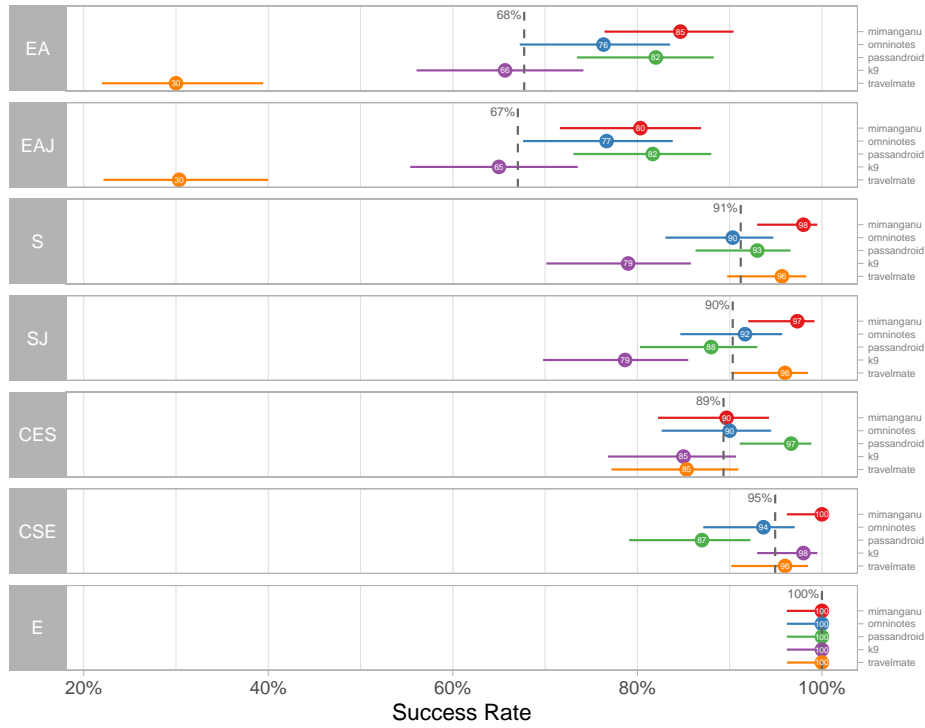


Figure 9: Average translation success rate for each test tool and app plotted with 95% confidence intervals. **EA** - EyeAutomate (Native), **EAJ** - EyeAutomate (Java), **S** - Sikuli (Native), **SJ** - Sikuli (Java), **CES** - Combined (EyeAutomate (Java) with Sikuli (Java) as backup), **CSE** - Combined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

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

977 Table 12 reports the results of the logistic regression. We observe that all
 978 3rd generation tools, except EAJ exhibit a significant difference (all p-values
 979 $< 10^{-3}$) in terms of success rate from the reference tool, i.e. EAJ. Moreover we
 980 can observe a significant difference among the apps.

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

986 The average success rate for SikuliX test cases was higher than 90%. Break-
 987 ing down the results by App, we observe peaks near 98% for the MiMangaNu
 988 app. As we can deduce by looking at the confidence interval, no significant
 989 difference could be found between the average success rate of scripted versions
 990 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
Appminnotes	-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

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

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

1000 The breakdown of the proportion of Passing, Flaky and Failing Tests, mea-
 1001 sured for the six sets of 3rd generation scripts and divided by app are reported
 1002 in fig. 10. We can observe that for all the five applications, a high percentage
 1003 of EyeAutomate test cases (both with the test scripts and through the Java
 1004 APIs) failed in all executions. This percentage reaches 70% for TravelMate. On
 1005 the other hand, test cases written with SikuliX showed no failing test cases for
 1006 MiMangaNu and a maximum 17% of failing test cases for K9-Mail.

1007 The usage of combined 3rd generation test cases led to even better results,
 1008 thanks to the usage of a backup visual tool when a recognition with the first
 1009 tool failed. While the combination with EyeAutomate as the primary tool had
 1010 a residual amount of failing test cases, the combination with SikuliX as primary
 1011 tool proved to have the lowest number of failing test cases overall: just a single
 1012 one for PassAndroid and TravelMate.

1013 In addition to reporting the success rate distributions, we also analyzed –
 1014 based on the different results – the individual test cases, to understand the
 1015 reasons that led some tools and test cases to fail.

1016 For instance, the EyeAutomate visual recognition library was unable to find
 1017 visual components like the Navigation Drawer icon (consisting of only three
 1018 white lines on a blue background, see figure 11), and the More Options icon
 1019 (consisting of three small dots, see figure 12)³. Some flakiness in SikuliX test

³Discussions 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.,

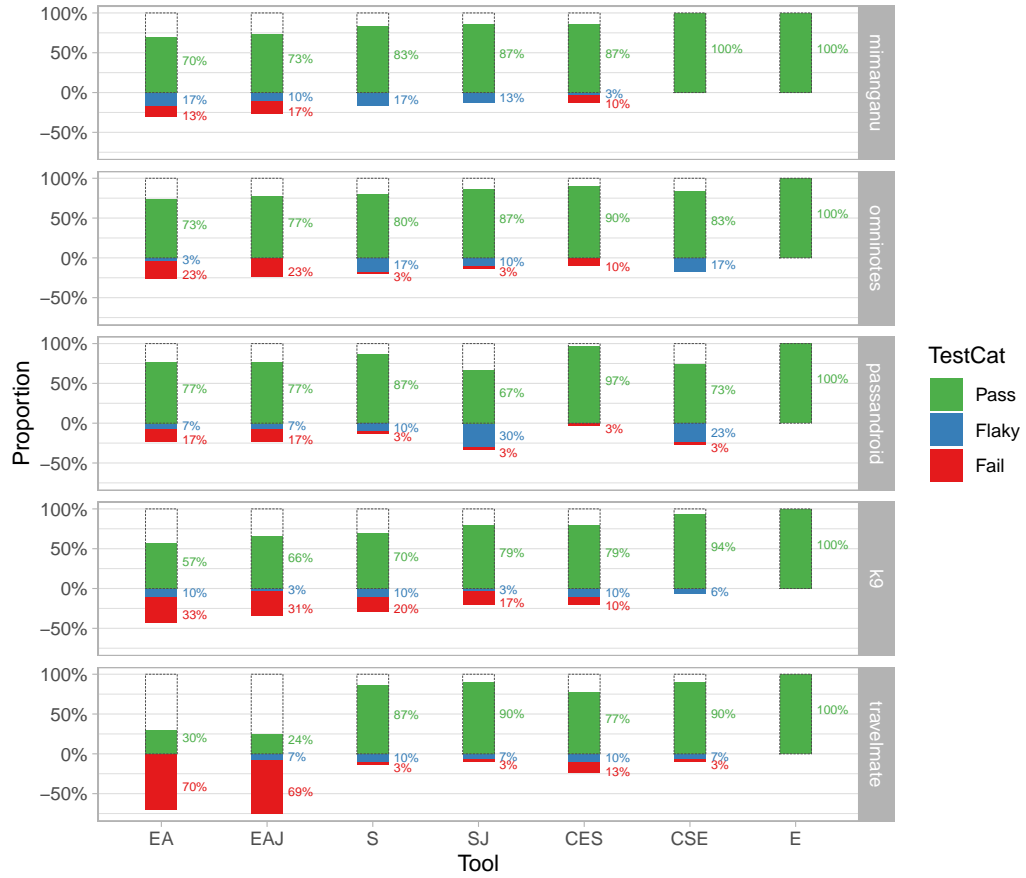


Figure 10: Proportion of passing, flaky and failing translated test cases



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
<i>Overall 1</i>	0.43	0.08

1020 cases was connected to the need for swipe operations, which were less precisely
 1021 reproduced⁴.

1022 The described failures showcase the varying capabilities of different image
 1023 recognition algorithms and also a secondary benefit of translation. Hence, trans-
 1024 lation can not just be used to transfer one generation of GUI tests to another,
 1025 but also allows translation to different technologies, or combinations of tech-
 1026 nologies, to best fit a certain context or purpose.

1027 Hence, the combination of the tools improves the overall success rate for all
 1028 apps. The CES combination had a residual number of failing test cases even
 1029 when all executions were passing with CSE. Those remaining failing test cases
 1030 may be justified with situations in which EyeAutomate executes an operation
 1031 on a wrong locator (i.e., a false positive of the image recognition engine), hence
 1032 deviating the test case from its correct execution. In contrast, when an EyeAu-
 1033 tomate test gets stuck for not recognizing a widget, using the image recognition
 1034 algorithm of SikuliX as a backup allows “runtime repair” of the test case without
 1035 moving to the wrong states of the GUI.

1036 Table 13 reports the average number of times the “backup” tool was used
 1037 in the test cases. The overall values confirm that the SikuliX tool proved more
 1038 robust, being used more often as a backup of a failing EyeAutomate locator
 1039 than the vice-versa.

1040 **Answer to RQ3:** None of the 3rd generation scripts achieved the same suc-
 cess 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.

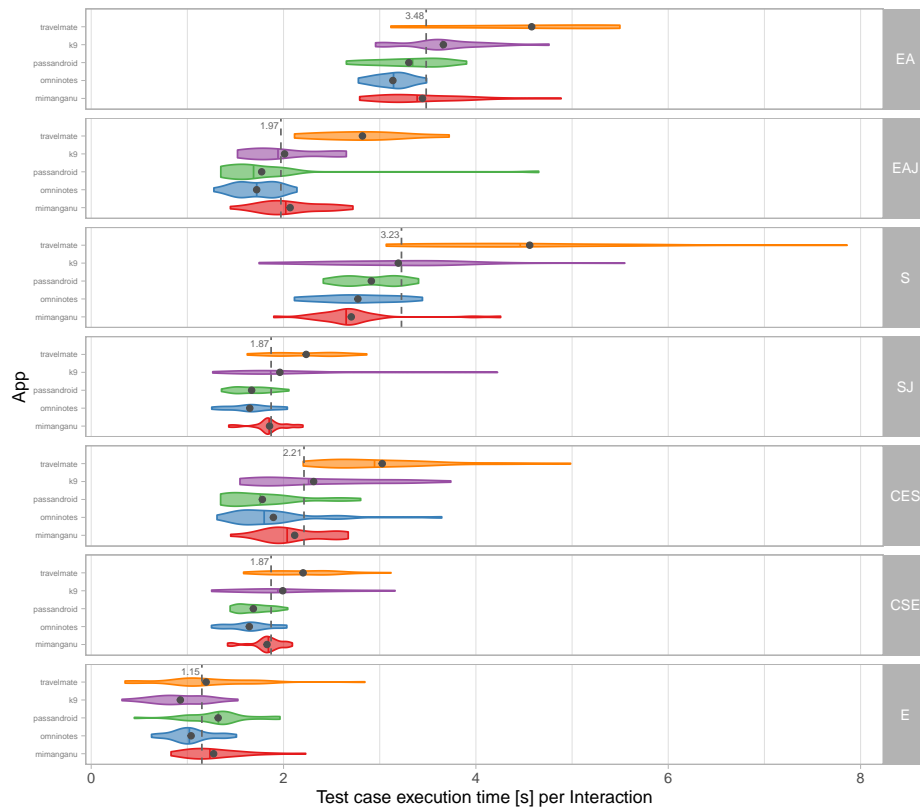


Figure 13: Distribution of execution time, normalized by number of interactions, by tool and app

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-omninoes	-0.196	< 0.001
App-passandroid	-0.114	< 0.001
App-k9	0.109	< 0.001
App-travelmate	0.702	< 0.001

1041 *5.4.4. RQ4 - Visual Scripts Performance*

1042 Figure 13 presents the test case execution time for each tool and app, nor-
 1043 malized by the number of interactions performed. Only the passing test case
 1044 executions were taken into consideration for the computation. Checks (either
 1045 of individual widgets or the full screen) were counted as interactions since the
 1046 time required by the image recognition algorithm to find a match is equivalent
 1047 regardless if the purpose is to identify a position for interaction or simply to
 1048 find if a widget is present. Once more, Espresso has been added as a benchmark
 1049 to see how the other tools compare. The number of interactions performed in
 1050 Espresso test cases was the same as in the translated 3rd generation ones, except
 1051 the final full check of the app screen (i.e. the assertion) that was not present in
 1052 developed Espresso test cases.

1053 Table 14 reports the coefficients for the linear regression of the time per
 1054 interaction vs. the indicator variables corresponding to the different tools and
 1055 apps. The non-parametric permutation test on the linear model coefficients
 1056 shows a significant difference between measured average time per interaction
 1057 depending on tool (all $p < 10^{-16}$) and a significant effect of the application (all
 1058 $p < 10^{-16}$). In other words, the results say that:

- 1059 1. changing the target tool of the translated scripts is sufficient to provide
 1060 a significant change in the measured time per interaction, due to varying
 1061 image recognition algorithms adopted;
- 1062 2. changing the AUT leads to a significant change in the measured time per
 1063 interaction, a reasonable result since different AUTs may need different
 1064 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.

⁴An 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

1065 The magnitude of average time variation induced by change of the tool is one
 1066 order of magnitude larger than switching to a different AUT.

1067 Espresso guaranteed a lower average execution time per interaction. The
 1068 main reason for this is the tool’s use of properties that has inherently higher
 1069 performance due to less required calculations than the image recognition ap-
 1070 proach. Additionally, Espresso, being integrated into the Android framework,
 1071 can filter the intents for Activity switching and automatically wait for the exact
 1072 time for an Activity or widget to appear on the screen, thus minimizing waiting
 1073 times. The higher execution time of 3rd generation tools is a finding that has
 1074 been reported by many works in the literature [48][10], and the results reported
 1075 in this paper are in line with manuscripts comparing the performance between
 1076 the two technologies. Similarly, the comparison between the execution time of
 1077 different 3rd generation tools is a supporting contribution of this study.

1078 The difference in average time per interaction caused by the different apps
 1079 can be explained by the fact that the patterns of interactions with the five ap-
 1080 plications are different, e.g., PassAndroid and TravelMate required more longer
 1081 swipe operations than the other AUTs.

1082 The fastest tools after Espresso were the Java version of SikuliX, and Eye-
 1083 Automate’s Java API. Hence, an interesting observation is that, both tools had
 1084 lower performance when tests were written in the tool’s specific syntax than in
 1085 their respective Java APIs. This may be explained by the fact that the test
 1086 cases were run inside a Java environment, instantiating script runners provided
 1087 by the respective libraries. An alternative explanation is that the script tools’
 1088 implementations caused additional overhead that is not present in when the
 1089 bare-bone image recognition libraries are used.

1090 As expected, the combined test versions had a bit worse performance than
 1091 any of the tools individually. The reason is TOGGLE’s approach of creating
 1092 tests that always try with one tool first, and only if it fails, after a set time (of 5
 1093 seconds), uses the second tool. Both combined solutions had, however, a better
 1094 performance than the scripts developed in the tool-specific syntaxes.

1095 Table 15 shows a comparison between the average interaction time for all
 1096 the tools, and the relative net interaction times (obtained by removing sleep
 1097 and backup times that had been inserted in the test cases).

1098 We measured a relevant difference also between the net interaction time re-
 1099 quired by the scripted versions of the tests compared to the test suites leveraging
 1100 the Java APIs of the two adopted testing tools. The difference is of more than
 1101 one second per interaction for both SikuliX and EyeAutomate. No substantial

1102 difference in terms of net interaction time, on the other hand, was found between
1103 the test suites written in Java, with CSE (Combined Sikuli-Eyeautomate) being
1104 the fastest and CES and EJ (Combined EyeAutomate-Sikuli and the Java API
1105 of EyeAutomate) the slowest.

1106 **Answer to RQ4:** The 2nd generation approach, as expected, has significantly
lower execution time compared to any of the single or combined 3rd generati-
on 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.

1107 5.4.5. RQ5 - Robustness to Device Fragmentation

1108 To evaluate the fragmentation fragility reduction, we utilized the combined
1109 Sikuli-Eyeautomate (CSE) test suites, obtained from the previous experiments,
1110 since they had the best overall behaviour in terms of success rate for all AUTs,
1111 on the Nexus 5X.

1112 The dumbbell plot in fig. 14 shows the success rates of the visual tests
1113 originally captured and converted on the Nexus 5X and executed in nine other
1114 different devices (bullet), versus the success rate of the suite – automatically –
1115 re-captured on the very same devices (triangle).

1116 We observe that the test cases translated on the Nexus 5X (bullets in fig. 14)
1117 were almost completely portable to the Nexus 6P and Pixel XL devices, likely
1118 because of the similar size of the pictorial rendering of the device on-screen.
1119 On the other hand, most likely due to rendering differences of varying pixel
1120 density, the test suite was not fully portable to the Nexus 5, even though it
1121 shared the screen size with the Nexus 5X. The portability was also limited on
1122 Nexus 6 and Pixel, which was caused by minor changes in the rendering of the
1123 buttons. For devices with smaller screens (Galaxy Nexus, Nexus 4, Nexus One,
1124 Nexus S), the tests could rarely be ported due to the very different sizes of the
1125 rendered widgets. On average, on all devices, only 31.6% of visual test cases
1126 were portable (less than 10% for five devices out of nine).

1127 These results clearly demonstrate that the negative impact of Device Frag-
1128 mentation on Visual tests is quite high for Android applications when the screen
1129 size and the pixel density of the target device are different from those of the
1130 device on which the test suite has been captured.

1131 On the other hand, looking at the success rate of the re-captured test suites
1132 (triangles in fig. 14), the vast majority of the test cases that were translated
1133 to specific devices, starting from a common layout-based counterpart, were suc-
1134 cessful (at most flaky). Two devices (the Nexus S and Nexus One) exhibited
1135 the lowest percentage of working translated test cases. This was caused by the
1136 fact that several Espresso test cases (3 for OmniNotes, 2 for PassAndroid, one
1137 for MiMangaNu, eight for K9-mail) were not executable on those devices. Due
1138 to their smaller screen size, different layouts were rendered, with widgets that
1139 were not displayed to the users. Whilst this was a hindering result for the ex-
1140 periment, it also showed a benefit of translation, since the 3rd generation tests

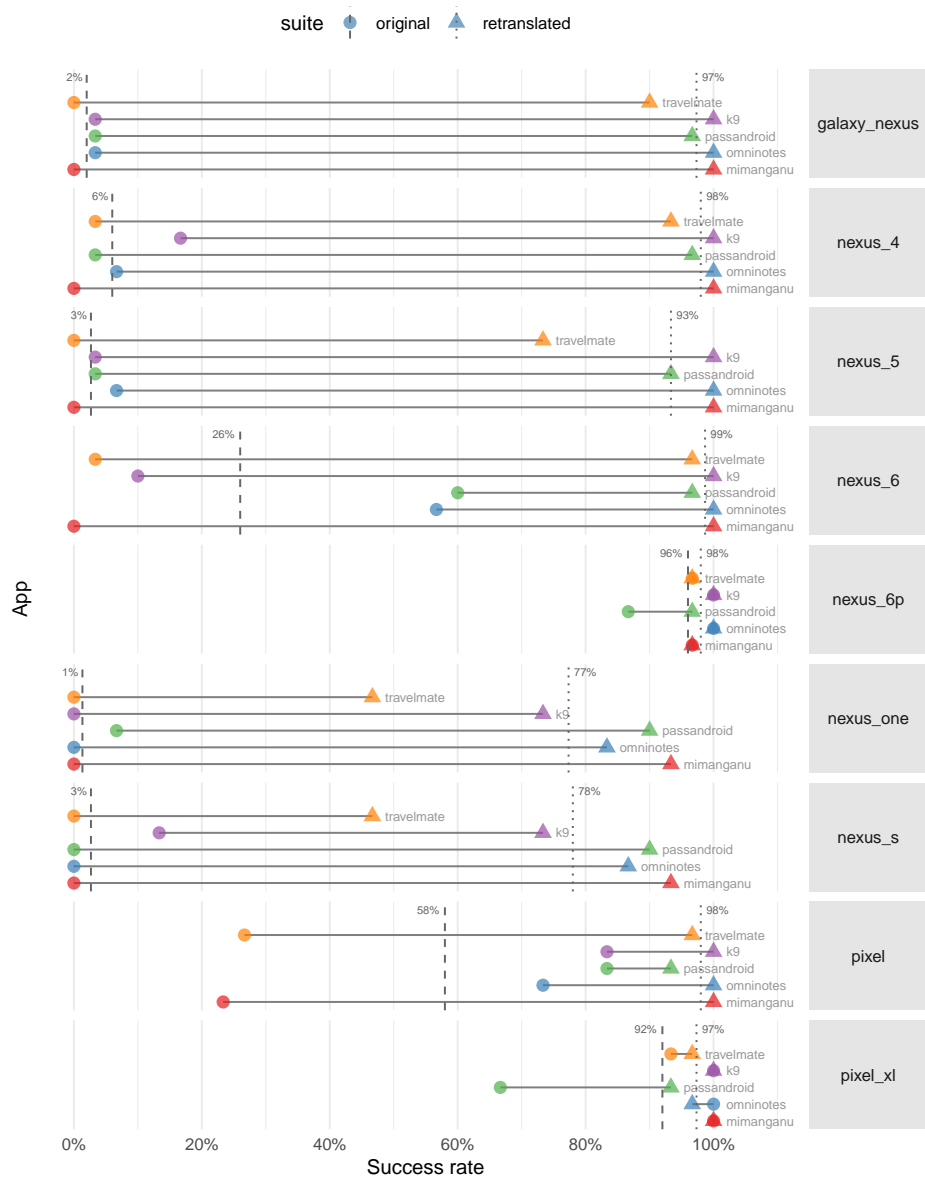


Figure 14: Change in visual test success rate between suite originally captured on different device and re-captured on same device.

1141 would fail due to layout issues. In fact, some of these faults could be classified
 1142 as being detrimental to app usability, i.e. tests of a non-functional attribute of
 1143 the AUTs.

1144 Also, the options button was not shown because a physical button – then
 1145 removed from Android devices – was used to that purpose. In these cases, the
 1146 layout-based test cases themselves were fragile to device diversity. Thus they
 1147 could not be used to create valid image recognition-based counterparts. For all
 1148 other devices, 90.0% or more of the translated test cases were passing or flaky.

1149 The described results suggest that it is possible to adapt existing layout-
 1150 based test suites to varying devices with minimal effort, to be spent in modifying
 1151 the residual visual locators or oracles that cause false negatives in the translated
 1152 3rd generation test cases.

1153 **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.

1154 5.4.6. RQ6 - Robustness to Graphic Fragility

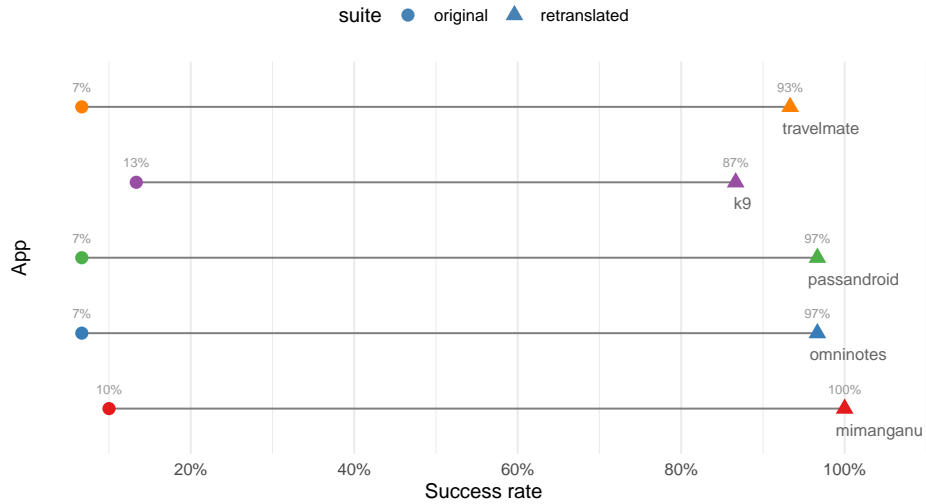


Figure 15: Percentage of passing (or flaky) test cases for the original test suites, and for the test suites re-translated after graphic modifications were applied to the apps

1155 The dumbbell plot in figure 15 report the results of the evaluation that we
 1156 performed to measure the robustness to Graphic Fragility of the translation-
 1157 based approach. We injected graphic modifications to 15 separated widgets
 1158 for each of the five software objects. In this case, we utilized the CSE output
 1159 combination of the TOGGLE tool as the test suite for our experiment.

1160 In the plot, we report the percentage of passing or at least flaky test cases for
1161 each app, for the test suite translated before modifications were injected (bullet),
1162 and for the one re-translated after the modifications were applied to the app
1163 (triangle). The discussed results can be considered as a proxy to evaluate the
1164 benefits of the application of a translation-based approach to cope with graphic
1165 maintenance of existing AUTs.

1166 It can be seen that the graphic changes invalidated the vast majority of the
1167 original 3rd generation test cases for all the considered apps. The number of
1168 original test cases that still passed on the modified AUTs ranged from 2 to 4
1169 (for K9-Mail).

1170 On the other hand, when the test cases were re-translated, the number of
1171 passing ones ranged from 26 (for K9-Mail) to 30 (for MiMangaNu).

1172 Some test cases did not pass even after the translation: for instance, the
1173 changes in OmniNotes involved a modification in a small TextView that likely
1174 was not recognized after the modification by both the 3rd generation drivers.
1175 For K9-Mail, three test cases were not re-translatable because of the Espresso
1176 tool itself not working properly on rotated views.

1177 However, the effort of repairing those test cases (for instance, by selecting
1178 a bigger portion of the screen instead of just the bounding box of the specific
1179 widget, or by recreating an interaction with a rotated widget) can be deemed
1180 minimal if compared to the manual re-capture of all the changed widgets, and
1181 the manual fixing of all the test cases using them.

1182 **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.

1183 6. Discussion

1184 The experiments we conducted have highlighted the feasibility and the bene-
1185 fits of a translation-based approach from 2nd generation to 3rd generation in the
1186 mobile testing domain.

1187 The migration of a layout-based test suite to a visual one lowers the need
1188 for costly manual operations – for both creation and maintenance – in any
1189 application domain. These costs are particularly high for mobile applications,
1190 where changes in the GUIs are frequent and where fragmentation issues (related
1191 to both graphical modifications and device change) have a significant impact.

1192 We implemented the proposed approach in a tool named TOGGLE. Al-
1193 though the tool covers a subset of interactions available in Espresso, it is capa-
1194 ble of translating the most commonly used ones – w.r.t. test suites developed
1195 with such tool – and this translation proved to be fast and correct for nearly all
1196 interactions.

1197 A thorough assessment of any tool requires the evaluation of the usefulness
1198 of its output. Therefore we evaluated two usage scenarios: (i) re-translation

1199 of the same test suite in case of fragility induced by device fragmentation; (ii)
1200 re-translation of the same test suite in case of graphical fragility. We observed
1201 the portability of test suites to different devices was enhanced by this approach;
1202 this would likely lead to reduced maintenance costs when the graphical features
1203 of the AUT are changed.

1204 As such, the results provide a proof of concept and indicate that the users
1205 of such a translation-based approach can get the benefits of visual testing, with
1206 a significant reduction of the cost for capturing the right oracles and locators.

1207 It is important to notice that the current approach assumes that the AUT
1208 is not faulty, i.e., that the 3rd generation test cases are obtained on a version
1209 of the application that has no regressions. If defects are encountered at the
1210 time of translation, wrong captures may be obtained for both visual locators
1211 and oracles, leading to erroneous sequences of interactions in the resulting visual
1212 test scripts, that would require manual effort from the testers to be repaired. We
1213 observe that such drawbacks are similar to those affecting model-based testing,
1214 i.e. the automated inference of models of the GUI from the AUT. This latter
1215 technique is significantly cheaper than the manual creation of models, however,
1216 it may produce faults in the generated models, hence requiring validation and
1217 additional information that has to be provided manually [49][50].

1218 6.1. Practical implications

1219 It is important to emphasize the consequences of our findings in the context
1220 of practical development:

- 1221 • *Suitability for automation*: the translation process is entirely automated;
1222 manual intervention in case of wrong translations, that in our experience
1223 affected less than 2% of test cases. We also need to stress that ours is a
1224 proof-of-concept tool, not an industrial-grade instrument.
- 1225 • *Translation efficiency*: the tool is able to translate 2nd generation test
1226 cases into 3rd generation ones at a pace of one every four seconds.
- 1227 • *Test dependability*: the resulting 3rd generation test cases were less depend-
1228 able than the 2nd generation counterparts. The combination of different
1229 image recognition algorithms improved that, but still the proportion of
1230 passing tests ranged between 73% to 100%. As a disclaimer, we observe
1231 that these are limitations inherent in 3rd generation tools and are not spe-
1232 cific to our approach. Therefore future improvements in this category of
1233 tools would trigger improvement in our approach too.
- 1234 • *Test execution*: the execution of the translated 3rd generation test cases
1235 took roughly 60% more time than the 2nd generation ones. As for de-
1236 pendability, here the approach is limited by inherent characteristics of the
1237 3rd generation tools.
- 1238 • *Device fragmentation reduction*: our approach was able to raise repro-
1239 ducibility of tests across different devices from 32% to 93%. This is not a
1240 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.

1284 7. Related Work

1285 The proposed approach adds to studies available in the literature, that con-
1286 ceptualize the possible benefits of a combined approach of 2nd and 3rd genera-
1287 tion testing tools [2]. The results of those empirical evaluations show that the
1288 2nd generation approach interacts and asserts the GUI model, leading to more
1289 false-negatives than 3rd generation approach for acceptance test; on the other
1290 hand, the 3rd generation approach, which mimics a real user’s interaction with
1291 the GUI, reports more false positives than the 2nd generation approach for sys-
1292 tem testing. The different behaviour and the different type of information that
1293 is verified against the actual state of the application suggest that the techniques
1294 should be adopted in combination for better test performance. 2nd and 3rd ge-
1295 neration testing tools have also been compared in terms of learnability, quality,
1296 and robustness of the developed test suites, as perceived by practitioners [44].

1297 The present work is also related to existing literature that aims at combining
1298 layout-based and visual testing, that evaluates the benefits and drawbacks of
1299 both techniques, or that proposes novel methodologies to generate more robust
1300 and portable visual locators.

1301 7.1. Translation-based approaches

1302 A translation-based approach similar to that used by TOGGLE has been
1303 already proposed by Leotta et al. in the field of Web-Application testing,
1304 where DOM-based 2nd generation test cases (developed with Selenium Web-
1305 Driver) were translated to 3rd generation test cases (written with Sikuli) [7][52].
1306 The reported evaluation of the tool highlighted the enhanced maintainability
1307 and ease of re-creation of 3rd generation test cases, compared to the original
1308 2nd generation ones from which they were obtained.

1309 The said approach is based on the translation of DOM-based test cases,
1310 that can be generalized to any web application, even web-based or hybrid mobile
1311 applications. The approach we propose is instead specifically tailored to Android
1312 apps since it is based on layout-based test cases which use native properties of
1313 Android apps as locators (e.g., unique ids or content descriptions).

1314 Our approach also covers a higher number of interactions with the SUT
1315 than PESTO, which instead only covers click, type and check instructions. This
1316 property is due to the higher number of instructions featured by the platform-
1317 specific tools considered as the source for the translation. The TOGGLE tool
1318 needed to be designed to manage commands that cannot be translated directly
1319 to atomic 3rd generation instructions, e.g., scroll and swipe operations.

1320 Compared with PESTO, our approach does not consider the possibility of
1321 interacting with multiple elements with the same on-screen appearance, even
1322 though this limitation can be solved by future developments of the project.

1323 On the other hand, while the test cases generated with PESTO required
1324 some (even though minimal) manual adaptation of the generated test code, our
1325 tool does not require any manual adaptation of the generated visual test scripts.

1326 The difference between the architectures required by PESTO and our tool
1327 underlines how – albeit being similar in concept – the translation of 2nd to

1328 3rd generation test cases entails different criticalities if applied to web-based or
1329 mobile test cases, and can be used as a technique to mitigate different issues
1330 that are domain-specific.

1331 7.2. Repair-based approaches

1332 Many studies in the literature have focused on a repair-based approach,
1333 aiming at correcting locators or instructions in test cases that fail when the
1334 AUT changes. Imtiaz et al. highlight the main trends in the field of studying
1335 test scripts repairing automation, applied on the web domain [53]. On the other
1336 hand, few tools were specific to the GUI testing of mobile applications. Li et al.
1337 introduce ATOM, a tool for automated maintenance of test scripts for mobile
1338 applications [54]. To perform this task, ATOM uses two different models: an
1339 *Event Sequence Model* (ESM) and a *Delta ESM* (DESM), that respectively
1340 represent a possible event sequence and the possible changes done on the GUI
1341 transitioning from a version of the application to the next one.

1342 CHATEM [3] extends ATOM to implement change-based testing. In prac-
1343 tice, taking two different versions of the same application (e.g., two consecutive
1344 releases), the tool can extract the changes between the two GUIs and to generate
1345 maintenance actions for each change, combining them to create repair actions
1346 for the broken test scripts.

1347 The described tools, compared with TOGGLE, are specifically tailored to
1348 solve the issue of (graphic) change-related fragility and need the extraction of a
1349 model of the user interface to enable the repair of broken test suites.

1350 7.3. Computer vision-based approaches

1351 Several studies have designed approaches based on computer vision to adapt
1352 test suites designed for a given SUT on different devices. Thereby, those studies
1353 aimed at reducing the costs for tackling the issue of fragmentation of visual
1354 test cases. Yu et al. described LIRAT [55], an image-driven tool that aims at
1355 recording and replay test scripts for the same application on different devices
1356 and platforms. The tool is based on image understanding techniques (namely,
1357 the SIFT feature extraction algorithm and KNN) to locate similar images on
1358 different renditions of the same GUI. Differently from the translation-based
1359 approach we propose, the tool does not take existing layout-based test cases
1360 as input, but instead relies on a single-step Script Recording phase performed
1361 by the tester/developer at the beginning of the process. Tuovenen et al. have
1362 described MAuto [56], a tool for the creation of cross-device visual test cases
1363 for Android apps. MAuto uses AKAZE features and is primarily tailored to
1364 reproduce user interaction with mobile games.

1365 Behrang and Orso described AppTestMigrator [57], a tool that attempts to
1366 automatically transform a sequence of events and oracles designed for a specific
1367 app to other similar applications. AppTestMigrator leverages commonalities
1368 between user interfaces to automatically migrate existing tests written for an
1369 app to another similar app.

1370 Cardenas et al. developed a tool named V2S [58] which generates replayable
1371 test scripts from video recordings of Android applications. The tool is primarily
1372 based on computer vision techniques.

1373 8. Threats to Validity

1374 **Threats to Construct Validity:** We have considered the success rate as
1375 a proxy for the evaluation of the precision of test cases, i.e., we expect that tests
1376 for working features must pass.

1377 The results about the performance of the generated 3rd generation test scripts
1378 are influenced by the static sleep instructions added during the translation of
1379 2nd generation scripts, which by converse need no explicit sleep instructions.
1380 The reports about the net time for interaction that are reported can just be
1381 used as an estimate of the lowest possible time for performing an interaction
1382 with the proposed Visual tools since a time interval for the rendering of the
1383 user interface, after the execution of commands, is not avoidable. In future
1384 enhancements of the tool, the sleep instructions should be made dynamic, uti-
1385 lizing GUI-state information to determine changes in the rendered screen before
1386 searching for visual locators. Dynamic sleep instructions are perceived to help
1387 the performance by mitigating unnecessary waiting time between consecutive
1388 interactions.

1389 **Threats to Conclusion Validity:** To verify the presence of a statistically
1390 significant difference among different target tools, we applied standard statistical
1391 tests. The results are clear cut and consistent with the visual representations
1392 that report standard (95%) confidence intervals or complete distributions.

1393 Researcher bias is another possible threat to the validity of this study since
1394 it involved a comparison in terms of different metrics of different 3rd generation
1395 testing tools. However, the authors have no reason to favour any particular
1396 approach, neither inclined to demonstrate any specific result.

1397 **Threats to External Validity:** We recognize that the documented exper-
1398 imental design includes some bias as only interactions supported by the trans-
1399 lator were used, i.e., only the Espresso commands belonging to the *OnView*
1400 family. The results of this evaluation are hence theoretically not generalizable
1401 to any Espresso test suite. However, TOGGLE's array of supported interactions
1402 include the most common ones used in Espresso, determined by an analysis of
1403 a set of 22,000 Espresso test files extracted from GitHub. Specifically, on the
1404 examined set, 97.33% of commands belonged to the *OnView* set, with just the
1405 remaining 2.67% belonging to the *OnData* set, not supported by the tool we
1406 developed. This finding indicates that our results would be applicable to the
1407 vast majority of test cases developed from open-source developers.

1408 We also performed a statistical analysis, to ensure that the set of properties
1409 and actions we used is representative of what is widely used in available GitHub
1410 repositories. Hence, we applied the Chi-Square tests to verify the Null Hypothe-
1411 ses $H_{0_{va}}$: *The View Actions in the developed test suites and the View Actions*
1412 *used in test cases mined from open-source repositories do not belong to the same*

1413 *distribution, and $H_{0_{vi}}$: The View Identifiers in the developed test cases and the*
1414 *View Actions used in test suites mined from open-source repositories do not be-*
1415 *long to the same distribution. We could reject both null hypotheses ($p < 10^{-16}$),*
1416 *hence we can assume that the View Actions and Interactions we used belonged*
1417 *to the same distribution of those in tests mined from open-source repositories.*

1418 The approach we developed is based on the assumption that the state of
1419 the application is always reflected by the pictorial GUI shown to the user. This
1420 means that 2nd generation test cases containing assertion on a lower level of
1421 abstraction (i.e., internal properties of the widgets or values of the variables
1422 declared in the code of Activities) cannot be entirely translated to equivalent
1423 3rd generation test cases, that cannot verify state changes that are not reflected
1424 by the appearance of the widgets of the graphical hierarchy. As well, Espresso
1425 test cases may contain direct interaction with methods declared in Activities
1426 without passing through widget interaction. These instructions do not have a
1427 visual testing counterpart. However, using these instructions in Espresso would
1428 result in developing unit tests of the SUT instead of pure GUI layout-based
1429 test cases, thereby having test artefacts that are by construct not eligible to be
1430 reproduced by visual test script drivers.

1431 As of now, the findings of the experimental section apply to the considered
1432 2nd and 3rd generation testing tools only, limiting the external validity of this
1433 work. However, it is possible to extend the syntaxes supported by TOGGLE by
1434 taking into consideration other testing tools, especially existing GUI Automa-
1435 tion Frameworks for Android (e.g., Appium, or UIAutomator). Additionally, it
1436 is not assured whether the precision, performance, and fragility reduction val-
1437 ues would be the same if measured with different typologies of applications with
1438 a very different graphical appearance compared to those that were considered
1439 (e.g., very graphically intensive projects such as games or video players). As
1440 well, the measured execution times proved to be strongly dependent on the type
1441 of interactions executed on the AUT, hence lowering the external validity of the
1442 results.

1443 9. Conclusion and Future Work

1444 In this work, we proposed the proof of concept of a novel approach for the
1445 creation of visual test cases in the mobile domain. The approach has been
1446 implemented in a tool called TOGGLE. The tool can translate layout-based
1447 2nd generation test cases, written in Espresso, to visual 3rd generation test cases
1448 using the SikuliX and EyeAutomate syntax. Similar approaches have previ-
1449 ously been evaluated in the field of web applications and DOM-based testing.
1450 However, to the best of our knowledge, this represents the first work in the
1451 literature about the translation-based generation of GUI test cases for mobile
1452 applications.

1453 To investigate the feasibility of the approach and its capability in overcom-
1454 ing known limitations of visual testing for mobile apps, we have experimented
1455 with five test suites that we developed for as many popular Android open-source

1456 applications. The tool was able to generate working test cases with high pre-
1457 cision and high success rate. It demonstrated that it is possible to reduce the
1458 testers' maintenance and development efforts by reusing existing layout-based
1459 test suites to create and maintain visual ones.

1460 In addition to fixing some current limitations of the tool at its current state
1461 of development, the natural prosecution of this work will be an evaluation of
1462 the approach in a real industrial environment, to quantify its benefits in the
1463 creation and maintenance of real-world test suites.

1464 As other future steps, we also identify the evaluation of an inverse translator,
1465 able to define layout-based test suites from existing visual ones. The backward
1466 translation would provide the added benefits of a possible creation of 2nd ge-
1467 neration test scripts through reuse of existing 3rd generation counterparts, and
1468 the mitigation of layout-based fragilities (i.e., changed 2nd generation locators
1469 invalidating layout-based test cases) by re-translation from 3rd generation tests
1470 that are still valid. This feature would allow a significant reduction of the
1471 maintenance cost of layout-based test suites, for which the impact of fragilities
1472 is known to be relevant [37].

1473 Also, we plan to provide companion translators, compatible with test scripts
1474 written with other layout-based testing tools, and to extend the approach to
1475 hybrid/web-based Android apps.

1476 References

- 1477 [1] I. Banerjee, B. Nguyen, V. Garousi, A. Memon, Graphical user interface
1478 (gui) testing: Systematic mapping and repository, *Information and Soft-
1479 ware Technology* 55 (10) (2013) 1679–1694.
- 1480 [2] E. Alégroth, Z. Gao, R. Oliveira, A. Memon, Conceptualization and evalu-
1481 ation of component-based testing unified with visual gui testing: an empir-
1482 ical study, in: *Software Testing, Verification and Validation (ICST)*, 2015
1483 IEEE 8th International Conference on, IEEE, 2015, pp. 1–10.
- 1484 [3] N. Chang, L. Wang, Y. Pei, S. K. Mondal, X. Li, Change-based test script
1485 maintenance for android apps, in: *2018 IEEE International Conference on
1486 Software Quality, Reliability and Security (QRS)*, IEEE, 2018, pp. 215–225.
- 1487 [4] L. Wei, Y. Liu, S.-C. Cheung, Taming android fragmentation: Character-
1488 izing and detecting compatibility issues for android apps, in: *Proceedings
1489 of the 31st IEEE/ACM International Conference on Automated Software
1490 Engineering*, 2016, pp. 226–237.
- 1491 [5] M. Kamran, J. Rashid, M. W. Nisar, Android fragmentation classification,
1492 causes, problems and solutions, *International Journal of Computer Science
1493 and Information Security* 14 (9) (2016) 992.
- 1494 [6] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, D. Lo, Under-
1495 standing the test automation culture of app developers, in: *2015 IEEE 8th*

- 1496 International Conference on Software Testing, Verification and Validation
1497 (ICST), 2015, pp. 1–10.
- 1498 [7] M. Leotta, A. Stocco, F. Ricca, P. Tonella, Pesto: Automated migration
1499 of dom-based web tests towards the visual approach, *Software Testing,*
1500 *Verification And Reliability* 28 (4) (2018) e1665.
- 1501 [8] R. Coppola, L. Ardito, M. Torchiano, M. Morisio, Mobile testing: New
1502 challenges and perceived difficulties from developers of the italian industry,
1503 *IT PROFESSIONAL (To appear)* 6.
- 1504 [9] L. Ardito, R. Coppola, M. Torchiano, E. Alégroth, Towards automated
1505 translation between generations of gui-based tests for mobile devices, in:
1506 *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops, ACM,*
1507 2018, pp. 46–53.
- 1508 [10] E. Alégroth, *Visual GUI Testing: Automating High-level Software Testing*
1509 *in Industrial Practice, Chalmers University of Technology, 2015.*
- 1510 [11] M. Linares-Vásquez, K. Moran, D. Poshyvanyk, Continuous, evolutionary
1511 and large-scale: A new perspective for automated mobile app testing, in:
1512 *Software Maintenance and Evolution (ICSME), 2017 IEEE International*
1513 *Conference on, IEEE, 2017, pp. 399–410.*
- 1514 [12] B. Sadeh, K. Ørbekk, M. M. Eide, N. C. Gjerde, T. A. Tønnesland,
1515 S. Gopalakrishnan, Towards unit testing of user interface code for android
1516 mobile applications, in: *International Conference on Software Engineering*
1517 *and Computer Systems, Springer, 2011, pp. 163–175.*
- 1518 [13] H. Zadgaonkar, *Robotium Automated Testing for Android, Packt Publish-*
1519 *ing Ltd, 2013.*
- 1520 [14] S. Negara, N. Esfahani, R. P. Buse, Practical android test recording with
1521 espresso test recorder, in: *Proceedings of the 41st International Conference*
1522 *on Software Engineering: Software Engineering in Practice, IEEE Press,*
1523 2019, pp. 193–202.
- 1524 [15] L. Gomez, I. Neamtiu, T. Azim, T. Millstein, Reran: Timing-and touch-
1525 sensitive record and replay for android, in: *Proceedings of the 2013 Inter-*
1526 *national Conference on Software Engineering, IEEE Press, 2013, pp. 72–81.*
- 1527 [16] Y. Hu, T. Azim, I. Neamtiu, Versatile yet lightweight record-
1528 and-replay for android, *SIGPLAN Not.* 50 (10) (2015) 349–366.
1529 doi:10.1145/2858965.2814320.
1530 URL <https://doi.org/10.1145/2858965.2814320>
- 1531 [17] M. Halpern, Y. Zhu, R. Peri, V. J. Reddi, Mosaic: cross-platform user-
1532 interaction record and replay for the fragmented android ecosystem, in:
1533 *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE In-*
1534 *ternational Symposium on, IEEE, 2015, pp. 215–224.*

- 1535 [18] M. Fazzini, E. N. d. A. Freitas, S. R. Choudhary, A. Orso, Barista: A tech-
1536 nique for recording, encoding, and running platform independent android
1537 tests, in: Software Testing, Verification and Validation (ICST), 2017 IEEE
1538 International Conference on, IEEE, 2017, pp. 149–160.
- 1539 [19] K. Moran, R. Bonett, C. Bernal-Cárdenas, B. Otten, D. Park, D. Poshy-
1540 vanyk, On-device bug reporting for android applications, in: Mobile Soft-
1541 ware Engineering and Systems (MOBILESoft), 2017 IEEE/ACM 4th In-
1542 ternational Conference on, IEEE, 2017, pp. 215–216.
- 1543 [20] K. Mao, M. Harman, Y. Jia, Sapienz: Multi-objective automated testing for
1544 android applications, in: Proceedings of the 25th International Symposium
1545 on Software Testing and Analysis, ACM, 2016, pp. 94–105.
- 1546 [21] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome,
1547 D. Poshyvanyk, Crashescope: A practical tool for automated testing of and-
1548 roid applications, in: Software Engineering Companion (ICSE-C), 2017
1549 IEEE/ACM 39th International Conference on, IEEE, 2017, pp. 15–18.
- 1550 [22] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, Z. Su,
1551 Guided, stochastic model-based gui testing of android apps, in: Proceedings
1552 of the 2017 11th Joint Meeting on Foundations of Software Engineering,
1553 ACM, 2017, pp. 245–256.
- 1554 [23] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, A. M. Memon,
1555 Mobiguitar: Automated model-based testing of mobile apps, IEEE software
1556 32 (5) (2015) 53–59.
- 1557 [24] T. Yeh, T.-H. Chang, R. C. Miller, Sikuli: using gui screenshots for search
1558 and automation, in: Proceedings of the 22nd annual ACM symposium on
1559 User interface software and technology, ACM, 2009, pp. 183–192.
- 1560 [25] E. Alégroth, M. Nass, H. H. Olsson, Jautomate: A tool for system-and
1561 acceptance-test automation, in: 2013 IEEE Sixth International Conference
1562 on Software Testing, Verification and Validation, IEEE, 2013, pp. 439–446.
- 1563 [26] E. Alégroth, A. Karlsson, A. Radway, Continuous integration and visual
1564 gui testing: Benefits and drawbacks in industrial practice, in: Software
1565 Testing, Verification and Validation (ICST), 2018 IEEE 11th International
1566 Conference on, IEEE, 2018, pp. 172–181.
- 1567 [27] E. Alégroth, R. Feldt, P. Kolström, Maintenance of automated test suites
1568 in industry: An empirical study on visual gui testing, Information and
1569 Software Technology 73 (2016) 66–80.
- 1570 [28] E. Alégroth, R. Feldt, On the long-term use of visual gui testing in indus-
1571 trial practice: a case study, Empirical Software Engineering 22 (6) (2017)
1572 2937–2971.

- 1573 [29] F. Dobsław, R. Feldt, D. Michaelsson, P. Haar, F. G. Neto, R. Torkar, Es-
1574 timating return on investment for gui test automation tools, arXiv preprint
1575 arXiv:1907.03475 (2019).
- 1576 [30] Y.-D. Lin, J. F. Rojas, E. T.-H. Chu, Y.-C. Lai, On the accuracy, effi-
1577 ciency, and reusability of automated test oracles for android devices, IEEE
1578 Transactions on Software Engineering 40 (10) (2014) 957–970.
- 1579 [31] M. Linares-Vásquez, C. Bernal-Cárdenas, K. Moran, D. Poshyvanyk, How
1580 do developers test android applications?, in: Software Maintenance and
1581 Evolution (ICSME), 2017 IEEE International Conference on, IEEE, 2017,
1582 pp. 613–622.
- 1583 [32] V. Garousi, M. Felderer, Developing, verifying, and maintaining high-
1584 quality automated test scripts, IEEE Software 33 (3) (2016) 68–75.
- 1585 [33] A. M. Memon, Automatically repairing event sequence-based gui test suites
1586 for regression testing, ACM Transactions on Software Engineering and
1587 Methodology (TOSEM) 18 (2) (2008) 4.
- 1588 [34] R. Coppola, M. Morisio, M. Torchiano, L. Ardito, Scripted gui testing
1589 of android open-source apps: evolution of test code and fragility causes,
1590 Empirical Software Engineering (2019) 1–44.
- 1591 [35] D. Han, C. Zhang, X. Fan, A. Hindle, K. Wong, E. Stroulia, Understand-
1592 ing android fragmentation with topic analysis of vendor-specific bugs, in:
1593 Reverse Engineering (WCRE), 2012 19th Working Conference on, IEEE,
1594 2012, pp. 83–92.
- 1595 [36] J.-H. Park, Y. B. Park, H. K. Ham, Fragmentation problem in android,
1596 in: 2013 International Conference on Information Science and Applications
1597 (ICISA), IEEE, 2013, pp. 1–2.
- 1598 [37] R. Coppola, M. Morisio, M. Torchiano, Mobile gui testing fragility: A
1599 study on open-source android applications, IEEE Transactions on Reliabil-
1600 ity (2018).
- 1601 [38] H. Zheng, D. Li, B. Liang, X. Zeng, W. Zheng, Y. Deng, W. Lam, W. Yang,
1602 T. Xie, Automated test input generation for android: towards getting there
1603 in an industrial case, in: Software Engineering: Software Engineering in
1604 Practice Track (ICSE-SEIP), 2017 IEEE/ACM 39th International Confer-
1605 ence on, IEEE, 2017, pp. 253–262.
- 1606 [39] Y. Liu, C. Xu, S.-C. Cheung, Characterizing and detecting performance
1607 bugs for smartphone applications, in: Proceedings of the 36th international
1608 conference on software engineering, 2014, pp. 1013–1024.
- 1609 [40] C. Wilke, C. Piechnick, S. Richly, G. Püschel, S. Götz, U. Abmann, Com-
1610 paring mobile applications’ energy consumption, in: Proceedings of the
1611 28th Annual ACM Symposium on Applied Computing, 2013, pp. 1177–
1612 1179.

- 1613 [41] A. Hovsepyan, R. Scandariato, W. Joosen, J. Walden, Software vulnera-
1614 bility prediction using text analysis techniques, in: Proceedings of the 4th
1615 international workshop on Security measurements and metrics, 2012, pp.
1616 7–10.
- 1617 [42] N. Mathur, S. A. Karre, Y. R. Reddy, Usability evaluation framework for
1618 mobile apps using code analysis, in: Proceedings of the 22nd International
1619 Conference on Evaluation and Assessment in Software Engineering 2018,
1620 ACM, 2018, pp. 187–192.
- 1621 [43] R. Feng, G. Meng, X. Xie, T. Su, Y. Liu, S.-W. Lin, Learning perfor-
1622 mance optimization from code changes for android apps, in: 2019 IEEE
1623 International Conference on Software Testing, Verification and Validation
1624 Workshops (ICSTW), IEEE, 2019, pp. 285–290.
- 1625 [44] L. Ardito, R. Coppola, M. Morisio, M. Torchiano, Espresso vs. eyeauto-
1626 mate: An experiment for the comparison of two generations of android
1627 gui testing, in: Proceedings of the Evaluation and Assessment on Software
1628 Engineering, ACM, 2019, pp. 13–22.
- 1629 [45] K. Srisopha, R. Alfayez, Software quality through the eyes of the end-
1630 user and static analysis tools: a study on android oss applications, in:
1631 Proceedings of the 1st International Workshop on Software Qualities and
1632 Their Dependencies, ACM, 2018, pp. 1–4.
- 1633 [46] J. Ferreira, A. C. Paiva, Android testing crawler, in: International Con-
1634 ference on the Quality of Information and Communications Technology,
1635 Springer, 2019, pp. 313–326.
- 1636 [47] R Core Team, R: A Language and Environment for Statistical Computing,
1637 R Foundation for Statistical Computing, Vienna, Austria (2018).
1638 URL <https://www.R-project.org/>
- 1639 [48] E. Borjesson, R. Feldt, Automated system testing using visual gui testing
1640 tools: A comparative study in industry, in: 2012 IEEE Fifth International
1641 Conference on Software Testing, Verification and Validation, IEEE, 2012,
1642 pp. 350–359.
- 1643 [49] C. Sacramento, A. C. Paiva, Web application model generation through
1644 reverse engineering and ui pattern inferring, in: 2014 9th International
1645 Conference on the Quality of Information and Communications Technology,
1646 IEEE, 2014, pp. 105–115.
- 1647 [50] W. Yang, M. R. Prasad, T. Xie, A grey-box approach for automated gui-
1648 model generation of mobile applications, in: International Conference on
1649 Fundamental Approaches to Software Engineering, Springer, 2013, pp. 250–
1650 265.

- 1651 [51] M. Leotta, A. Stocco, F. Ricca, P. Tonella, Using multi-locators to increase
1652 the robustness of web test cases, in: 2015 IEEE 8th International Confer-
1653 ence on Software Testing, Verification and Validation (ICST), IEEE, 2015,
1654 pp. 1–10.
- 1655 [52] A. Stocco, M. Leotta, F. Ricca, P. Tonella, Pesto: A tool for migrating
1656 dom-based to visual web tests, in: 2014 IEEE 14th International Working
1657 Conference on Source Code Analysis and Manipulation, IEEE, 2014, pp.
1658 65–70.
- 1659 [53] J. Imtiaz, S. Sherin, M. U. Khan, M. Z. Iqbal, A systematic literature
1660 review of test breakage prevention and repair techniques, *Information and
1661 Software Technology* 113 (2019) 1–19.
- 1662 [54] X. Li, N. Chang, Y. Wang, H. Huang, Y. Pei, L. Wang, X. Li, Atom:
1663 Automatic maintenance of gui test scripts for evolving mobile applications,
1664 in: 2017 IEEE International Conference on Software Testing, Verification
1665 and Validation (ICST), IEEE, 2017, pp. 161–171.
- 1666 [55] S. Yu, C. Fang, Y. Feng, W. Zhao, Z. Chen, Lirat: Layout and image recog-
1667 nition driving automated mobile testing of cross-platform, in: 2019 34th
1668 IEEE/ACM International Conference on Automated Software Engineering
1669 (ASE), IEEE, 2019, pp. 1066–1069.
- 1670 [56] J. Tuovenen, M. Oussalah, P. Kostakos, Mauto: Automatic mobile game
1671 testing tool using image-matching based approach, *The Computer Games
1672 Journal* 8 (3-4) (2019) 215–239.
- 1673 [57] F. Behrang, A. Orso, Test migration between mobile apps with similar
1674 functionality, in: 2019 34th IEEE/ACM International Conference on Au-
1675 tomated Software Engineering (ASE), IEEE, 2019, pp. 54–65.
- 1676 [58] C. Bernal-Cárdenas, N. Cooper, K. Moran, O. Chaparro, A. Marcus,
1677 D. Poshyvanyk, Translating video recordings of mobile app usages into
1678 replayable scenarios, in: *Proc. of 42nd Int. Conf. on Software Engineering*,
1679 2020, p. 13. doi:10.1145/ 3377811.3380328.

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 <i>img</i> ii. Type [BACKSPACE] (<i>arg1</i> times)	i. click(<i>img</i>) ii. type(Key.BACKSPACE) (<i>arg1</i> times)
click	i. Click <i>img</i>	i. click(<i>img</i>)
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 <i>img</i> i. Click <i>img</i> ii. Type <i>arg1</i>	i. hover(<i>img</i>) 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 <i>img</i> ii. MouseLeftPress iii. Sleep 500 iv. MouseLeftRelease	i. hover(<i>img</i>) ii. mouseDown(Button.LEFT) iii. sleep(0.5) iv. mouseUp(Button.LEFT)
typetext	i. Click <i>img</i> ii. Type <i>arg1</i>	i. click(<i>img</i>) ii. type(<i>arg2</i>)
openactionbarmenu	i. Type [CTRL_PRESS] ii. Sleep 10 iii. Type <i>m</i> iv. Sleep 10 v. Type [CTRL_RELEASE]	i. keyDown(Key.CTRL) ii. sleep(0.01) iii. type(<i>m</i>) 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 <i>arg1</i>	i. type(<i>arg1</i>)
pressmenukey	i. Type [CTRL_PRESS] ii. Sleep 10 iii. Type <i>h</i> iv. Sleep 10 v. Type [CTRL_RELEASE]	i. keyDown(Key.CTRL) ii. sleep(0.01) iii. type(<i>h</i>) iv. sleep(0.01) v. keyUp(Key.CTRL)
replacetext	i. Click <i>img</i> ii. Type [BACKSPACE] (<i>arg1</i> times) iii. Type <i>arg2</i>	i. click(<i>img</i>) ii. type(Key.BACKSPACE) (<i>arg1</i> times) iii. type(<i>arg2</i>)
swipedown ⁵	i. Move <i>img</i> ii. Sleep 10 iii. MouseLeftPress iv. MoveRelative "0" "250" v. MouseLeftRelease	i. r = find(<i>img</i>) 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)

⁵For 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.