Towards Automated Translation between Generations of GUI-based Tests for Mobile Devices

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

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

ABSTRACT

Market demands for faster delivery and higher software quality are progressively becoming more stringent. A key hindrance for software companies to meet such demands is how to test the software due to to the intrinsic costs of development, maintenance and evolution of testware. Especially since testware should be defined, and aligned, with all layers of system under test (SUT), including all graphical user interface (GUI) abstraction levels. These levels can be tested with different generations of GUI-based test approaches, where 2nd generation, or Layout-based, tests leverage GUI properties and 3rd generation, or Visual, tests make use of image recognition. The two approaches provide different benefits and drawbacks and are seldom used together because of the aforementioned costs, despite growing academic evidence of the complementary benefits.

In this work we propose the proof of concept of a novel two-step translation approach for Android GUI testing that we aim to implement, where a translator first creates a technology independent script with actions and elements of the GUI, and then translates it to a script with the syntax chosen by the user. The approach enables users to translate Layout-based to Visual scripts and vice versa, to gain the benefits (e.g. robustness, speed and ability to emulate the user) of both generations, whilst minimizing the drawbacks (e.g. development and maintenance costs). We outline our approach from a technical perspective, discuss some of the key challenges with the realization of our approach, evaluate the feasibility and the advantages provided by our approach on an open-source Android application, and discuss the potential industrial impact of this work.

ACM Reference Format:
Luca Ardito, Riccardo Coppola, Marco Torchiano and Emil Alégroth. 2018. Towards Automated Translation between Generations of GUI-based Tests for Mobile Devices. In *(ISSTA Companion/ECOOP Companion'18), July 16–21, 2018, Amsterdam, Netherlands.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3236454.3236504

ISSTA Companion/ECOOP Companion'18 ,
July 16–21, 2018, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5939-9/18/07…$15.00
https://doi.org/10.1145/3236454.3236488

1 INTRODUCTION AND RELATED WORK

Android (and, in general, mobile) applications are rapidly and progressively reaching higher levels of complexity in their user interfaces to provide the features required by users. Hence, thorough testing of mobile app Graphical User Interfaces (GUI) – emulating most of the user interaction and app functionalities – should be a well-established practice among mobile developers.

To test any application provided with a GUI, three main categories of automated approaches exist: 1st generation testing tools (also called Coordinate-based) identify elements of the user interfaces using the exact coordinates at which they are shown on screen; 2nd generation testing tools (also called Layout-based or Property-based) identify elements according to the definition of the screens of the app, and/or one or more of their properties (e.g. unique identifiers in the layout, text that they contain, etc.); 3rd generation testing tools (also called Visual GUI testing tools [1]) identify elements through image recognition algorithms, which capture the actual appearance of the elements as displayed to the user.

For Android apps, the behaviour and the appearance of the GUI screens are defined by components named *Activities* (i.e., Java classes where the functionalities of the screens are defined and callbacks are assigned to elements of the interface) and by Xml layout files that describe the properties and arrangement of the widgets. Even though the widgets may also be defined dynamically via code, screens are typically populated at once by the Activities, loading a layout file. Alternative layout files can be provided for the same screen, to take into account different characteristics of the displays on which the app can be run (e.g., screen size, pixel density, portrait or landscape orientation). Layout-based testing tools usually leverage information that can be extracted from .xml layout files to identify the elements of the user interface; being layout files available in the .apk package of any Android application, the knowledge of information from layout files allows to perform black box testing of Android interfaces when the source code of the Activities is not available. Visual GUI testing tools, on the other hand, are completely agnostic of the description of the user interface, basing all the interaction that they perform on the GUI on the on-screen recognition of images. Hence, they are used to provide black box system level tests, which exercise and evaluate the features of the apps at the abstraction of the GUI only.

However, although a variety of testing tools are available for Android testing, including GUI-based tools of both generations, evidence from literature and industrial practice suggests that these apps are not as deeply tested as their desktop counterparts. Instead, most Android developers just rely on manual testing [5, 13] and

ad-hoc testing, or totally neglect testing [10]. A key reason to this lack of testing relates to domain-specific challenges associated with mobile devices that adversely affect the ease with which they can be tested through the GUI compared to desktop systems. For instance, the quantity of different context events and gestures to which apps must respond, the diversity of devices on which apps are deployed, the tight coupling between back-end and front-end functionality and the high pace of evolution of mobile app appearances all contribute to make the test cases more fragile [9]. Fragility manifests as incorrectly failing tests on new releases of the app (and thus failed regression tests) due to even trivial changes in the apps' core functionality, control scheme or its visual interface definition [5].

As such, lack of testing has prominent, diverse and significant effects on mobile app development. Those effects mandate investigation of new approaches to ease the efforts associated with test script creation, maintenance and evolution. We therefore propose a combined use of Layout-based and Visual GUI test scripts, with the novel complement of an automated mechanism to translate between them. This approach will allow the techniques to leverage their individual benefits to counter their drawbacks and allow reuse of existing test scripts to test other layers of abstraction of the GUIs, raise coverage and reduce cost. A combined technique would also – as we discuss in the Motivation section – help overcome many of the core difficulties of testing mobile apps.

2 MOTIVATION

Although Layout-based testing tools are commonly used in industrial practice, e.g. Appium and Robotium [4], Visual GUI testing tools, e.g. Sikuli [16] and EyeAutomate[1] [3], are much less common despite academic evidence for their applicability, feasibility and usefulness [1]. Reasons include the lack of robustness and performance of Visual GUI testing compared to Layout-based testing [12]. However, in contrast, Layout-based testing cannot fully emulate the human user as interactions through GUI properties do not verify the system's appearance as shown to the human user. Thus, most research aimed at comparing the two techniques have concluded that a hybrid approach is required [2] where both generations are developed and used in parallel by the practitioner.

Instead of defining a hybrid approach, requiring the tester/developer to have knowledge about both generations of test automation, we propose an automated translation of test scripts from one generation to the other, through systematic reuse of test logic, coupled with translation of property-based locators to visual locators, and vice-versa. The tool would allow the tester/developer to focus on manually developing the test suite in one methodology only, and automatically generate the counterpart.

The proposal can hence provide benefits such as:

(1) Automated generation of Layout-based tests from Visual GUI tests and vice versa, reducing development costs.
(2) Reduced script maintenance costs through automated analysis of failing locators, and repair based on the other generation's scripts.
(3) Reuse of existing Layout-based tests for testing the SUT's visual appearance.
(4) Porting of visual scripts for different devices/configurations through strategic reuse of Layout-based tests, i.e. a single test can be run on a set of emulated devices to automatically obtain Visual GUI test scripts specific to each device.
(5) Limited need for costly manual, repetitive, and error-prone regression testing [11].

These benefits can mitigate challenges with Visual GUI test scripts such as their sensitivity to visual changes (e.g. size, resolution, color, etc.), whilst also mitigating challenges with Layout-based test scripts that are sensitive to code changes (e.g. id, tags, components, etc.). This is possible since Visual GUI test scripts operate against what the user sees and can emulate the user, albeit quite slowly compared to other automation techniques. In contrast, Layout-based test scripts execute more robustly by using GUI component properties, not emulating the user, but at a higher execution speed.

The respective robustness to changes in the definition of the layouts and in pure visual changes theoretically allows translation of valid test scripts from one test generation to the other, e.g. when the latter is broken due to fragility issues. This allows reduction of the cost of maintenance and repair of test cases throughout different versions of the same application, especially if they are used to perform regression testing. A single unavoidable case of fragility which would not be repaired by the application of the translation approach would be the simultaneous presence of changes in the graphic appearance and in the layout properties for the same elements of the user interface: in that case, both generations of test cases would fail, hence making the repair of one of the two impossible based on the other.

Several tools are already available for capturing the operations performed on an emulated Android device, and generate test scripts in a specific layout-based testing framework accordingly. For instance, the tool Barista [7] allows to capture, encode and run test scripts on different platforms. The tool leverages the official Espresso GUI automation framework[2]. The Android Studio IDE also offers an Espresso Test Recorder feature, which allows to create Espresso test scripts through the Capture & Replay technique. As well, existing literature has explored the possibility of automatically repairing test scripts, without performing translation steps to other abstractions of the GUIs: the works by Memon [14] and Zhang et al. [17] moved towards this direction. Those approaches, however, do not provide – as our proposal would – a visual testing counterpart, and hence are not able to check the real appearance of the application, and to recover from fragilities in the layout of the screens if properties are changed after the tests are generated.

To sum up, the core idea of this work is to combine the strengths of scripts of both generations to mitigate the weaknesses of the two approaches taken individually. In section 4, a motivating example is described, to underline which weaknesses of test scripts of both generations can be mitigated leveraging a translation process.

3 PROOF OF CONCEPT

The core idea behind the proposed translator is to use information provided by the scripts of one generation to create scripts for the other. This translation is split into two parts. First, the scenario

[1]EyeAutomate was previously known as JAutomate.

[2]https://developer.android.com/training/testing/espresso/

– a series of GUI actions – is obtained through an execution and examination of the test script, written with a testing tool of a given generation. Second, the GUI objects are identified for the destination syntax, based on the information available in the test scripts of the starting generation.

In practice, Layout-based information (IDs, coordinates, strings) should be extracted only using images and Visual GUI testing images be acquired using Layout-based information.

For instance, the translation of a Layout-based test to a Visual GUI test requires, at run-time, the X- and Y-coordinates of a GUI component together its width and height to take a screen capture. The screen captured image, together with the action performed in the script, can then be represented in a syntax suitable for a Visual GUI testing tool to replicate the action with computer vision.

Similarly, a Visual GUI test script, during runtime, could return the position, width and height (at least approximately) of a GUI-object. This information could either be used to generate a completely new, partially completed, Layout-based GUI component, or use heuristics to look up if an existing component matches the coordinate, width and height information. Together with the action performed in the Visual GUI test script, this information could then be translated to the specific syntax of a new Layout-based test script.

3.1 Architecture of Layout-based to Visual GUI test scripts translator

The translation process from Layout-based to Visual GUI test scripts starts from a single script or a suite of scripts created with a property-based testing tool.

The process can be described with the following set of separate logic blocks, as shown in figure 1.

- The *Parser* first performs a static analysis of the code of layout files of an application's packages, and perform enhancements that may help in reducing fragilities of test cases. In this phase, all user interface elements that are used in tests are assigned – if they do not possess any – unique ids by modifying the fields of the layouts they are defined in. This modification allows elements to be uniquely identified on the screen and can be used for creation of new test scripts, without ambiguities. Other improvements can be performed on production code, e.g. the definition of String constants when hardcoded text is identified in the definition of widgets, to ease the recognition of text and the repairing of failing test cases.
  The parser then enriches the original Layout-based test code with instructions that allow the extraction – after each operation that is performed on the GUI – of information that is then needed for the translation, e.g. commands to take screen captures of the current activity, to trace clicks and coordinates, and calls to image manipulation libraries to create the individual images on which Visual GUI test scripts will be based.
- The *Runner* executes the enhanced Layout-based test script on an instrumented Android Virtual Device; at runtime, for each operation performed on the GUI, visual data is collected from the device, e.g. the appearance of button where an action has been performed. The outcome of the execution phase is a trace of all the performed actions in the form of tuples consisting of *operation type*, *interacted image* (screenshot), and layout-based information.
  The Runner also checks the outcome of the original Layout-based test: if the test triggers any exception the developer is signalled and the translation process is aborted. In the proposed architecture, the Runner block is independent from both the chosen Layout-based and Visual GUI testing tools. Additionally, this step encodes extracted images with suitable id-numbers to preserve traceability to the original Layout-based test script. We also propose that said imagery be saved together with other meta-data, such as coordinate information, to ease identification in the translation of other test cases (i.e., without having to generate screen captures for widgets already found in previous test scripts).
- Finally, the *Test Case Generator*, based on the trace of operations and screen captures given as output by the previous module, translates the set to a 3rd generation Visual test script, according to the chosen output scripting language (e.g., text files written in the EyeAutomate syntax).
  The new generated test script is then merged with existing 3rd generation test scripts; it can be used immediately for the test of the actual appearance of the app as shown to the user, or be part of a test suite for future regression testing.

3.2 Architecture of Visual to Layout-based GUI test scripts translator

The translation process from Visual to Layout-based GUI test scripts (Figure 2) starts from a single script or a suite of scripts created with a Visual GUI testing tool.

The process can be described with the following set of separate logic blocks, as shown in figure 2.

- In the first phase, *Instrumentation* is performed of the Android Project. This step is necessary for the translation and to go from visual reference to test code. The instrumentation of the Android Project enables (or modifies already existing) callbacks to any widget shown on the screen. The callbacks contain code that logs the interactions that have been performed on the GUI of the tested app during execution of a visual test script. Future evaluations of the features offered by the available tools (like UI Automator viewer) may make the Instrumentation of the Android Project unnecessary, if it is found that sufficient information about the user interface is obtainable without the need for adding callbacks in the app code.
- The *Executor* module runs the Visual GUI tests on the emulated device on the desktop screen. At each step of the test case, as in a typical execution of a Visual GUI test script, the position of the image on screen and its coordinates are identified. Together with height and width information from the expected image, additional information can be acquired about the interacted element by cross-referencing available information with either Layout-based data or other meta-data. For instance, this can be done leveraging the debug

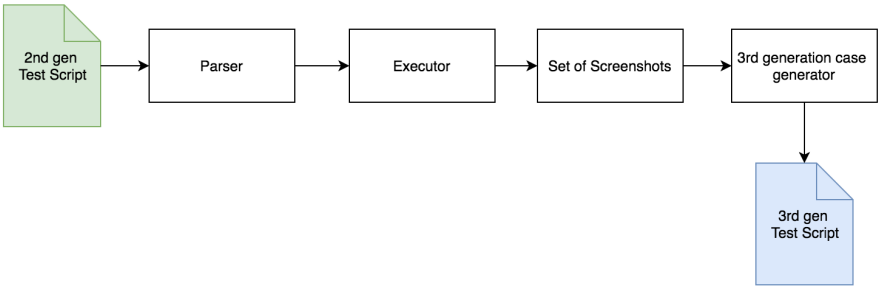


Figure 1: Architecture of translator from Layout-based to Visual GUI test scripts

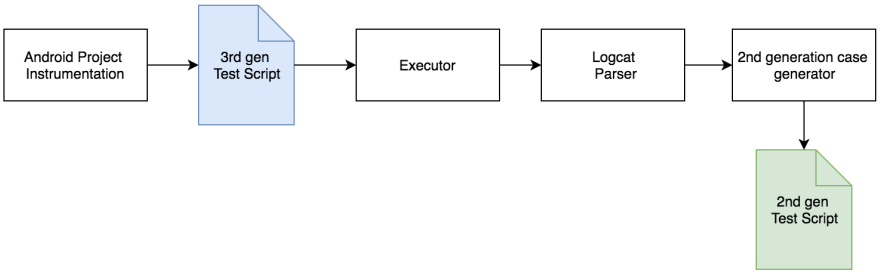


Figure 2: Architecture of translator from Visual to Layout-based GUI testing tools

connection with the instrumented device, e.g. using ADB (i.e., Android Debug Bridge) and the Android UI Automator Viewer, which allows navigation of an XML-description of a dump of the current interface shown on the emulated device screen. The extracted data can thereby be used to correlate an element represented by the Visual GUI testing tool as only an image with a Layout-based element represented by a set of properties.

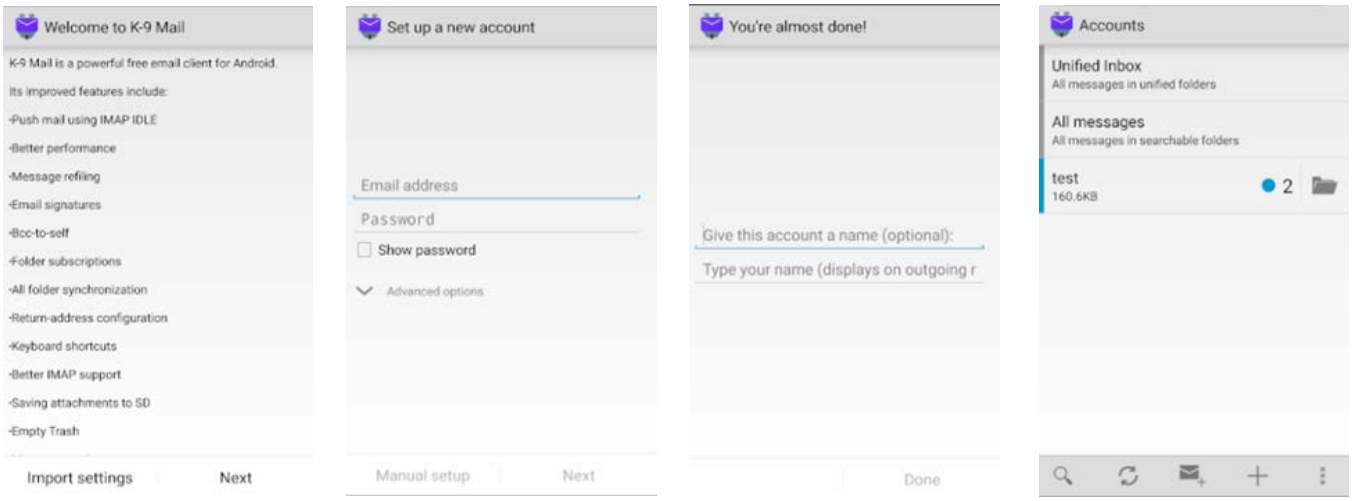
Since the operation of dumping the current screen may be long and requires the UI to remain still, the Executor may need to insert sleep instructions between consecutive operations in the Visual GUI test scripts. Additionally, the transition of certain UI elements might require additional steps to be inserted into the test scripts. For instance, for drop-down lists, Layout-based tools generally access the elements directly without expanding the lists. In contrast, Visual GUI testing tools must first expand the list to make the elements visible to be able to interact with them.

Due to the high abstraction of Visual GUI testing scripts, a proposed technical solution to ease and speed up the translation to Layout-based scripts is to store additional meta-data about existing objects from previously run test scripts (e.g., coordinates, properties, actual appearance). This may enable association of images in new test scripts with already interacted objects (it is the case, for instance, of buttons that are interacted in two different test cases). Additionally, the meta-data must be aligned with the Layout-based test data to ensure 1-to-1 association between Layout-based and Visual elements used in the test cases.

Step	Screen	Widget Description	Operation
1	s1	Next Button	Click
2	s2	Email account Form	Type test account email
3	s2	Password Form	Type test password
4	s2	Next Button	Click
5	s3	Account description Form	Type test account description
6	s3	Account name Form	Type test account name
7	s3	Done Button	Click
8	s4	Activity Title	Check that "Accounts" is shown
9	s4	Account List Item	Check that test name is shown

Table 1: Steps for Authentication use case of K-9 mail

- The output of the Executor logic block is a trace of the operations that will compose the translated test script: a log of tuples with the *properties* associated with an identified Visual element, and the *action* performed on the element. This information is given as output through the built-in Logcat tool of Android. The *Logcat Parser* logic block of the translator is in charge of parsing such trace, in order to obtain a language and technology-independent sequence of operations and widget descriptions that can be then used for the generation of test scripts.
- Finally, the *Test Case Generator*, based on the output of the previous module, creates Layout-based test script in the syntax desired by the user. The generated test script is then merged with existing test cases to be replayed as part of the Layout-based generation test suite counterpart.



(a) **s1 - WelcomeMessage Activity**

(b) **s2 - AccountSetupBasics Activity**

(c) **s3 - AccountSetupNames Activity**

(d) **s4 - WelcomeMessage Activity**

Figure 3: Screens and Activities traversed by the authentication use case

4 APPROACH VALIDATION

The purpose of this section is to provide a preliminary validation of our proof of concept, describing the benefits that would result from the use of a combined approach, based on mutual translation between Layout-based and Visual GUI test scripts, for Android application testing. Since much of this research is still in a planning phase, only a few components of the translator are completed. Thus, much of the proof of concept study was performed using manual identification and translation of components and actions. Hence, operations that would otherwise have been performed by the automated translator that we envision.

As our sample AUT, similar to a previous case study we conducted to understand the types of fragilities exposed by Scripted Android GUI testing [6], we have chosen K-9 mail, a popular and long-lived (366 releases) open-source e-mail client whose source code is available on GitHub[3]. The version on which our investigations are based is V5.500-SNAPSHOT. In our discussion, we call this original version of the application *v1*, as it is cloned from the GitHub repository. We have developed a sample test case to exercise the authentication feature of the application with two tools pertaining to different generations. All the tests have been run on an Android Virtual Device, namely a Nexus 5 with Android API 24 installed, with enabled device frame and hardware keyboard inputs.

When it is first launched, without any account already registered, the K-9 Mail app provides a wizard to perform a registration of a user e-mail account and to set up its name and description. Such use case for the registration of an account is described by the sequence of operations shown in table 1: for every operation to be performed, the table reports the screen on which it has to be performed, an high-level description of the element, the type of operation to perform. The four screens of the app that are traversed are shown in figure 3, with the respective Activity names indicated in the captions.

4.1 Test scripts definition for original release

As our reference Layout-based testing tool, we have chosen UI Automator[4], a GUI testing framework developed by Android to test app interfaces as well as the operating system's one. The UI Automator API allows to abstract elements of the user interface that are visible on the current screen of the device as UIObjects, and to retrieve them upon searches performed on properties like their description, identifier, contained text. The API also allows to manage more complex types of widgets, like scrollable elements or collections, with dedicated classes.

We developed the UI Automator test script for the Authentication use case using the Android Studio IDE, and inside the K-9 Mail application project, having full access to the AUT production code and the .xml layout files describing its user interface. Resource IDs used in the UI Automator test script have been collected launching the application and using the UI Automator Viewer tool, retrieving the "resource-id" field for any of the widgets to be interacted. Table 2 shows the retrieved ids for the elements interacted throughout the use case. Executed on *v1*, the test script runs to completion, as it is shown by the JUnit result provided by the Android Studio IDE.

As our reference Visual testing tool, we have chosen EyeAutomate[5], a tool leveraging image recognition to automate interaction on any kind of GUI software. To test Android applications, the tool requires the AUT to be launched on a virtual device, rendered on the screen of a desktop pc on which the Visual GUI testing tool is launched.

[3]https://github.com/k9mail/k-9

[4]https://developer.android.com/training/testing/ui-automator.html
[5]http://eyeautomate.com/index.html

Step	Object Description	Object ID
1	Next Button	next
2	Email account Form	account_email
3	Password Form	account_password
4	Next Button	next
5	Account description Form	account_description
6	Account name Form	account_name
7	Done Button	done
8	Activity Title	action_bar_title_first
9	Account List Item	description

Table 2: Retrieved IDs for UIAutomator test case

Step	Reference Image
1	
2	
3	
4	
5	
6	
7	
8	
9	

Table 3: Retrieved images for the EyeAutomate test script

We have developed the EyeAutomate test script using the companion Visual Script Editor EyeStudio[6]. Reference images, shown in table 3, have been gathered from a first manual execution of the application, leveraging the image capture tool embedded in the EyeStudio suite. Interaction points with the captured image have been fixed in all cases to the center of the identified images. Sleep instructions have also been inserted in the test script at the beginning of the test case, and at every screen transition: this prevents the visual testing tool to search too early for elements that have not yet been rendered on the virtual device screen, and thus fail. Executed on v1, the test script runs to completion, as it is shown by the .html result file provided by the EyeStudio suite.

4.2 Layout-based fragility induction

To highlight the fragility of layout-based test scripts to modifications performed on the properties of the interacted widgets, we performed a simple modification on the definition of a layout of *v1* of the AUT, thus obtaining the version that we called *v2a*. In particular, as it is shown by figure 4, the resource ID associated to the *Done* button is changed from *"done"* to *"completed"*, in the layout file (namely, wizard_done.xml) where the button is declared.

In the UI Automator test script, the Resource IDs are inserted as constants; if the layout file is modified outside the development IDE, or no automated refactoring functions propagate the modifications on the widget property, the test script will still search for the old widget ID even though it has changed. In particular, the UI Automator test script fails (launching the UIObjectNotFoundException) when the following line of code is reached.

```
UIObject done_button = new UiObject(new UiSelector()
```

[6]http://eyeautomate.com/eyestudio.html

```
<Button
    android:id="@+id/done"
    style="?android:attr/buttonBarButtonStyle"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_weight="1"
    android:background="@drawable/selectable_item_background"
    android:text="@string/done action" />
```

Layout excerpt from v1

```
<Button
    android:id="@+id/completed"
    style="?android:attr/buttonBarButtonStyle"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_weight="1"
    android:background="@drawable/selectable_item_background"
    android:text="@string/done_action" />
```

Layout excerpt from v2a

Figure 4: Modification in the layout file between v1 and v2a

```
.resourceId("com.fsck.k9.debug:id/done"));
```

In all transitions like the one from *v1* to *v2a*, containing only modifications in the widget definition and properties, all test cases leveraging widgets with varied properties may fail; in the provided example, the test case fails due to an ID change fragility. On the other hand, the Visual test runs to completion without errors.

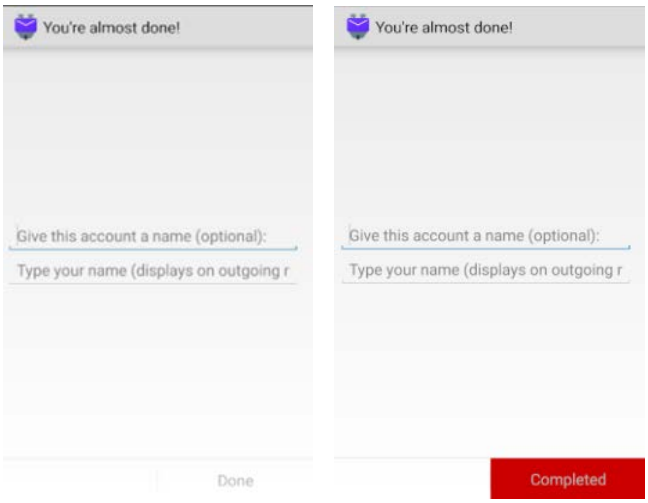
In the translation approach that we envision, an automated feature should be launched at the beginning to instrument the application to perform additional logging operations, using the Android Development Bridge to log all the interactions that are performed with the widgets of the user interface, along with their IDs. This way, the visual test case - which is still valid - can be used to retrieve the actual (changed) id of the *Done* button when it is clicked, generating a valid companion layout-based test case.

4.3 Visual fragility induction

To cause a fragility in the visual test script, we performed a couple of graphic modifications on the original version *v1*, leading to the version that we called *v2b*. In particular, we changed the appearance of Screen 3, modifying the background color and the text of the *"Done"* button. Each of the two modifications performed would be sufficient, if applied alone, to invalidate a visual test case. The changed appearance of the screen between *v1* and *v2b* is shown in figure 5.

Without any modification in layout and widget definitions, there is no impact in scripted test cases, since the button to be clicked is unambiguously retrieved by its unchanged ID. On the other hand, the visual recognition test case that we developed experiences a failure, due to the inability to identify an element with the appearance of the Done button, still linked in the test script to the screen capture shown in table 3.

In the translator approach that we envision, the layout-based test script is properly enhanced, with the addition of instructions to log – at any interaction performed – the type of interaction, the absolute coordinates where it has took place, and a capture of



(a) **Screen 3 in v1**

(b) **Screen 3 in v2b**

Figure 5: Graphic change in Screen 3 between v1 and v2b

the current screen. Using the coordinates of the interaction, and possible additional information extracted from layout files (e.g., size and padding of the interacted widget) a picture of the new appearance of the interacted widget can be cut from the capture of the whole current screen, when the working Layout-based script is executed.

This way, the translator can automatically create a valid companion visual script for the layout-based test script, without any manual maintenance intervention from the tester/developer.

5 NEXT STEPS

As the next step of our current work, we plan to progressively implement the components of both the translators, from Layout-based to Visual GUI test scripts and vice versa. The translator will be initially specialized for Android, and implemented as a plug-in for the Android Studio IDE. Along with the modules of the architecture shown in Section 3, we plan to develop some helper modules that will be needed to uniform the declaration of widgets in layout files, and to make all elements of a GUI unambiguously identifiable for the creation of a Layout-based test script: a Parser module for .xml files will be developed, with the objective of creating a map of the IDs used by an application, and assigning a new unique ID to all widgets not having it.

As a second step, we plan to develop our translator as a standalone tool, to leverage possible Enhancer and Test Case Generator modules for test languages that are not specific to mobile applications. This will be generalized through the two-step translation procedure described in Sections 3.1 and 3.2, where tuples of components and interaction types are first extracted independently of scripting language. Such tool-independent, general representation of the interactions with a GUI element, is then in a second step translated to a language of the user's choice. This approach aims to enhance the generalizability of the developed tool and to make it extendable to work also with other Layout-based testing tools (e.g., Selenium [12] for web application testing).

As pointed out in section 3, the step of identifying layouts and properties from clicked buttons on a GUI is the most complex to carry out, hence the described architecture and algorithm may be subject to limited variations. Also, further investigations are needed to understand the feasibility of the creation and translation of textual oracles from Layout-based to Visual GUI test scripts and vice-versa: an investigation of the possibility of using OCR techniques to "read" textual contents from screen captures, to use them as parameters for layout-based scripts construction, is also planned. Additionally, careful investigations are needed for the possibility of translating correctly complex gestures (e.g., swipe operations) from layout-based test scripts, where they are rendered with simple function calls, to visual-based test scripts, where they would need the combined use of multiple atomic instructions.

The evaluation of our translator approach will also look into possible solutions for solving challenges where translation from one generation to another is simply not possible. For instance, when information is visually rendered, e.g. in a graph, which can be tested with the Visual GUI test scripts, but not accessed by the Layout-based testing tools due to interface restrictions. Additionally, careful investigation is needed for the possibility of translating correctly complex gestures (e.g., swipe operations) from layout-based test scripts, where they are rendered with simple function calls, to visual-based test scripts, where they would need the combined use of multiple atomic instructions.

To validate our project, we plan to apply the translator to sets of open-source Android applications, to evaluate the impact on the needed effort to develop test cases with a proper coverage and on the fragility of test suites. We also plan to measure the gain – in terms of effort and time required – obtainable by using the translation approach for creating a new visual GUI test suite from an existing scripted one, and vice-versa. Additionally, we aim to evaluate the combined Layout-based and Visual GUI testing procedure in an industrial environment, to assess the applicability and usefulness of such an approach in a real scenario. This will also enable an evaluation of the cost savings and potential return on investment (ROI) of the approach compared to, for instance, currently used manual practices.

As future work, after the completion and validation of the translator between the two generation of testing frameworks, we plan to extend the approach to other forms of testing (e.g., for web applications) which may suffer of the same kinds of fragilities.

6 CONTRIBUTION AND INDUSTRIAL IMPACT

GUI-based testing is growing in commonality and importance in industrial practice due to the growing adoption of continuous ways-of-working [15]. As a result, more companies have realized the need for multi-layered automated testing. However, maintenance of GUI-based tests is costly since the tests are affected by changes on all levels of abstraction beneath the GUI. This cost greatly affects the tests' longevity and companies ability to sustain multiple test suites over time, with negative effects on the software system quality, time to market and, as a result, customer satisfaction.

The proposed approach could thereby provide a significant contribution as it enables companies to expand their testing capabilities at very low cost. This capability is given by the ability to translate existing test cases from one generation to another. In practice, this means that companies that use Appium [4], Selenium [12] or other Layout-based tools that are commonly used in industry, can quickly acquire Visual GUI tests for regression testing. A single translation can have long-term effects, as it enables automated maintenance of the tests. Hence, if the GUI is changed, the Layout-based tests can be used to maintain the Visual GUI test suite, and vice versa if the code-base but not visual appearance is changed. Thus providing unprecedented capabilities for testware reuse and cost-savings.

Additionally, due to its positive impact on quality feedback, the approach is perceived an enabler for more stringent continuous ways-of-working practices such as continuous deployment. The reason is that the translator enables robust automated GUI-based testing that mitigates the need for manual testing, lowering development lead-times and quickening time to release. Additionally, as the approach alleviates the need for manual regression testing, it opens the possibility of more exploratory testing [8], which enables efficient identification of new faults.

7 CONCLUSION

Automated GUI-based testing is becoming more commonly used in practice, not to mention in the ever expanding mobile app domain where GUI-functionality is key to market success. Several tools are available; they can be categorized as 2nd generation (Layout-based or Property-based) or 3rd generation (Visual, or Image-recognition based). Both approaches are able to replace manual GUI-based testing, but suffer from high development and maintenance costs and are therefore seldom used together. This is a challenge as both techniques have mutually exclusive benefits and drawbacks related to factors such as speed, robustness and ability to emulate the human user, which make the two generations approaches complementary to one another.

In this work we proposed a novel approach for automated translation between one generation of tests to the other. This approach would enable gaining the benefits of both generations whilst mitigating the costs and drawbacks of the individual approaches; The result would be a reduced need for manual testing, shorter lead-times, and enabling practices such as continuous deployment and more exploratory testing.

Consequently, the approach would help software development organizations to meet the software market's growing needs for faster delivery and higher quality software.

REFERENCES

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

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

[3] Emil Alégroth, Michel Nass, and Helena H Olsson. 2013. JAutomate: A tool for system-and acceptance-test automation. In *Software testing, verification and validation (icst), 2013 ieee sixth international conference on*. IEEE, 439–446.

[4] Haneen Anjum, Muhammad Imran Babar, Muhammad Jehanzeb, Maham Khan, Saima Chaudhry, Summiyah Sultana, Zainab Shahid, Furkh Zeshan, and Shahid Nazir Bhatti. 2017. A Comparative Analysis of Quality Assurance of Mobile Applications using Automated Testing Tools. *INTERNATIONAL JOURNAL OF ADVANCED COMPUTER SCIENCE AND APPLICATIONS* 8, 7 (2017), 249–255.

[5] Riccardo Coppola, Maurizio Morisio, and Marco Torchiano. 2017. Scripted GUI Testing of Android Apps: A Study on Diffusion, Evolution and Fragility. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*. ACM, 22–32.

[6] Riccardo Coppola, Emanuele Raffero, and Marco Torchiano. 2016. Automated mobile UI test fragility: an exploratory assessment study on Android. In *Proceedings of the 2nd International Workshop on User Interface Test Automation*. ACM, 11–20.

[7] Mattia Fazzini, Eduardo Noronha De A Freitas, Shauvik Roy Choudhary, and Alessandro Orso. 2017. Barista: A technique for recording, encoding, and running platform independent android tests. In *Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on*. IEEE, 149–160.

[8] Juha Itkonen and Kristian Rautiainen. 2005. Exploratory testing: a multiple case study. In *Empirical Software Engineering, 2005. 2005 International Symposium on*. IEEE, 10–pp.

[9] Thomas W Knych and Ashwin Baliga. 2014. Android application development and testability. In *Proceedings of the 1st International Conference on Mobile Software Engineering and Systems*. ACM, 37–40.

[10] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo. 2015. Understanding the Test Automation Culture of App Developers. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. 1–10. https://doi.org/10.1109/ICST.2015.7102609

[11] Martin Kropp and Pamela Morales. 2010. Automated GUI testing on the Android platform. *Testing Software and Systems* (2010), 67.

[12] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. 2014. Visual vs. DOM-based web locators: An empirical study. In *International Conference on Web Engineering*. Springer, 322–340.

[13] Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Kevin Moran, and Denys Poshyvanyk. 2017. How do developers test android applications?. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE, 613–622.

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

[15] Daniel Ståhl and Jan Bosch. 2014. Modeling continuous integration practice differences in industry software development. *Journal of Systems and Software* 87 (2014), 48–59.

[16] Tom Yeh, Tsung-Hsiang Chang, and Robert C Miller. 2009. Sikuli: using GUI screenshots for search and automation. In *Proceedings of the 22nd annual ACM symposium on User interface software and technology*. ACM, 183–192.

[17] Sai Zhang, Hao Lü, and Michael D Ernst. 2013. Automatically repairing broken workflows for evolving GUI applications. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM, 45–55.