AUTOMATING TEST SCRIPTS FOR ANDROID UI TESTING

Jessica Turner and Judy Bowen

University of Waikato, Tauranga, New Zealand

ABSTRACT

The Espresso capture/replay testing tool for Android applications creates tests that are prone to test fragility, in that when small changes occur to the user interface, tests break and are unable to be re-run. To reduce the fragility and its impact inherent in Espresso tests, we take a model-driven development approach to test generation. Using interaction sequence models as the basis for generation, we are able to create test scripts that can be run in Android Studio identically to manually recorded tests. This process simplifies scripts when compared with those generated by recording and reduces the time required by developers to create and maintain the test suite, resulting in higher quality testing and validation of Android user interfaces.

KEYWORDS

Capture/Replay Testing, Android, Model-driven development, User interfaces

1. INTRODUCTION

The Espresso tool is one of the most popular testing frameworks used in smart phone application (app) development for Android. One of the reasons for this is its inclusion in Android Studio [1] the integrated development environment (IDE) for building Android apps. Espresso is a capture/replay testing tool, which means developers can record user interactions and replay these as tests, removing the need to write complex test scripts in order to test user interactions. The recorder generates test scripts in either the Kotlin or Java Programming languages (the same languages used to create Android applications in Android Studio) and these are used to replay the tests. Stack traces and other usual output, such as pass/failure of tests, is provided to allow developers to debug the UI behaviours.

Despite the many benefits the Espresso tool provides, similar to other capture/replay tools it is prone to test fragility (when tests break and are unable to be re-used). This occurs because the test scripts generated are tightly coupled with the UI and even small changes to the UI can, therefore, result in tests breaking. This requires further work for developers and increases development and testing time as user interactions must be manually recorded again in order to restore the test suite.
Model-driven development relies on creating models of a system in order to generate code and/or tests. Models can be created using the system requirements for both the front-end and back-end of applications. The benefits of using a model-driven development approach lies in both the time saved implementing the application and also in verification and validation of the system behaviours. The models act as a design artefact that clearly specify the code and its structure resulting in less debugging and error fixing. When changes occur in the system, rather than updating the implementation directly, using a model-driven approach allows us to regenerate code and/or tests. This results in software that is easier to maintain.

David C. Wyld et al. (Eds): SEC, NLPML, CYBI, SIGL, CRIS, AIAPP, CoSIT- 2024
pp. 01-13, 2024. CS & IT - CSCP 2024 DOI: 10.5121/csit.2024.140701

In this paper we propose taking a model-driven approach to Android testing which will allow developers to regenerate test scripts when changes occur to the implementation. This will address the fragility present in tests currently produced by the Espresso testing tool while keeping the benefits of the tool's script generation. The model-based approach will allow developers to focus on refactoring the app and UI code rather than rewriting tests. This will, in turn, reduce time and effort in the development and testing process.

The primary contribution of this paper is a novel testing process using model-driven development to automatically generate capture/replay tests in the Espresso tool for Android applications. We also introduce the Espresso Generator tool which implements the test generation process. The rest of this paper is structured as follows: the following section gives more background context on existing tools and approaches for testing the UI of Android applications; in section 3 we explore the way in which capture/replay tests are created for Espresso while in section 4 we describe the models used to inform the test generation process. We then introduce the Espresso Generator which implements the model-driven testing process and explore tests for a "real-world" application. Lastly, we discuss the tool and its limitations, finishing with concluding remarks and future work.

2. BACKGROUND

Capture/replay testing is a technique whereby user interactions are stored as a script which can subsequently be played back to recreate the sequence of actions. Capture/replay tools can be divided into three main categories: recording, scripting or generation approaches. This defines how the tests are created. In a recording approach, a user interacts with the software under test (SUT) and their actions are recorded to create a replayable test script. Assertions can also be added at defined points in the recording sequence to check the state of the UI or expected values. Examples of such tools for Android include the Espresso Test Recorder, in-built into Android Studio [2] and GIFdroid, which allows users to provide developers with visual bug reports so they can replay execution traces for debugging purposes [3]. However, recording tests manually can be time consuming as it requires scenarios of use to be created and then a user to manually interact with the system. While the recorder will produce a test script that can be replayed repeatedly throughout the development process, the resulting test scripts are prone to fragility, depending on the script generation process used [4]. For example, if the UI is refactored, recorded test scripts are often unable to successfully reproduce sequences of interactions recorded using earlier versions of the UI.

A scripting-only process requires a tester to define a test script in a specified programming language. Again, this requires the development of test scenarios which must then be converted into replayable test scripts. The scripting approach requires detailed understanding of the SUT and its code, in order to define the methods required to interact with the UI elements. Behaviour-driven development approaches, such as the Cucumber tool [5], help to address this by generating test stubs which can be later fully written by testers, however this still requires some understanding of the SUT implementation. Most capture/replay tools provide both recording and scripting options, allowing the tester to use either method, or a combination of both. In Android Studio, testers are able to write test scripts for Espresso in either the Kotlin or Java programming languages, tests can then be replayed as per the recording approach.

A generation approach, such as the one we propose, uses some form of model or specification in order to automate the creation of test scripts. This removes the need for manual test recording or script writing. It also addresses the fragility often inherent in capture/replay tests. If the UI changes, tests can be re-generated from the model rather than having to be re-created which requires more manual effort.

Model-based testing (MBT) approaches rely on a model or specification of the SUT. The model describes the intended behaviours of the SUT and can be used to generate tests, create test oracles and ultimately determine whether or not the SUT behaviour matches the modelled behaviour. Many different approaches for MBT of interactive systems have been proposed, focusing on different types of models such as task models [6], design models [7], directed graphs of interaction sequences [8] etc. Less attention has been given to MBT for mobile devices, with many of the approaches that have been developed focussing on lightweight methods for model discovery. An example of model discovery approaches are seen in work by Amalfitano et al. who proposed using a GUIRipper tool which dynamically explores the UI of an Android application and generates event-based tests [9]. Cao et al. provide a refinement to dynamic model generation using their CrawlDroid tool to group widgets and apply priorities to guide exploration during the model building and increase efficiency [10]. A similar model-based approach is seen in [11] where the authors use dynamic exploration of an Android app to create a model and use this as the basis for mutation testing. When compared with state-of-the-art existing testing tools they found their approach discovered more unique defects/bugs than the existing tools.

One weakness of the model-based approaches described above, which use dynamic exploration, is that the models created are primarily event-based only. These enable test sequences of interactions to be created, but not assertions, as the model does not include information about functional changes to state or values that occur during the navigation. This is the motivation for the modelling and testing approach we propose in this paper, as it enables test sequences which incorporate both navigation and assertions. Rather than generating the models from the SUT, we assume a model-based development process whereby models of the system are created during the design process. Such models are typically richer than those which are dynamically generated as they include details of functionality beyond the interface and interactions and therefore support the generation of test oracles and/or assertions.

While creating models manually, rather than using dynamic ‘ripping’ approaches as described above is more time-consuming, the overhead of creating the models is offset by the automation of parts of the testing process and reduced design and maintenance costs. In addition, work by Di Martino et al. which explores the differences between automated test tools for Android with those manually generated to evaluate their effectiveness [12] found that human reasoning alongside automated tools produced the highest quality tests. This highlights the importance of automation to provide assistance to the testing process but also the need for human evaluation in order to produce meaningful tests with high test coverage. This informs the approach we propose here as we use automation to generate tests alongside human reasoning to create meaningful models of the UI.

In the next section we describe the use of the Espresso capture/replay tool which can be integrated into Android Studio and then demonstrate how interaction sequence models can combine the benefits of automation alongside human reasoning to provide a more robust test-generation method.

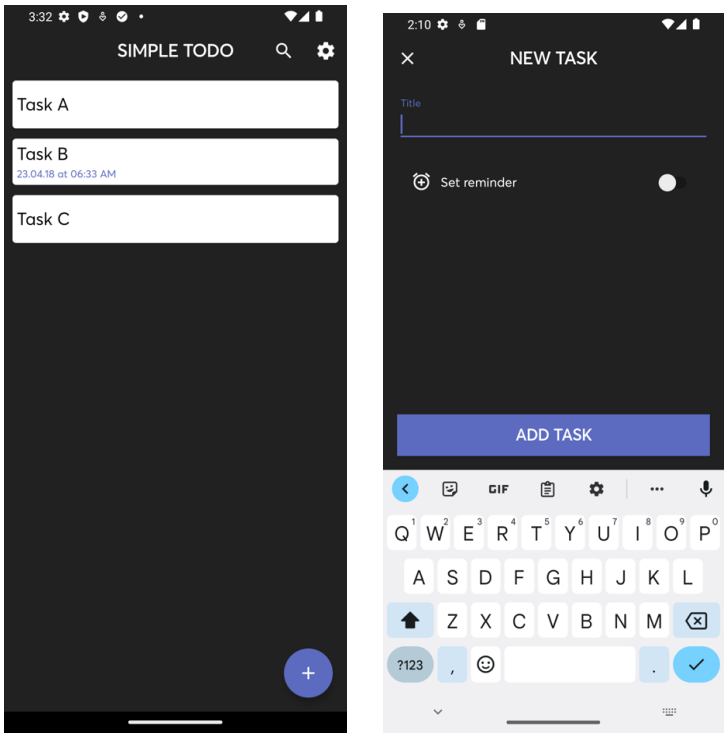
3. CAPTURE/REPLAY TESTING USING ESPRESSO

Espresso[1] is a testing tool implemented as part of Android Studio[2] that allows developers to write or record interaction sequences to create capture/replay test scripts for their Android applications. The scripts can be written in both the Kotlin and Java programming languages, which are the

[1] See https://developer.Android.com/training/testing/espresso
[2] See https://developer.Android.com/studio

primary languages for Android development, or they may be manually recorded using the Espresso recorder. The recorder will produce test scripts in the developer's preferred development language. The Espresso tool fits under both the recording and scripting categories, and exhibits the fragility problems outlined in the previous section.

In Android, interactive widgets are referred to as Views or Objects which exist inside an Android Layout. These widgets have unique properties which allows Espresso to recall which widgets to interact with as a test is run. Views or Objects are identified using matchers that determine a unique way of selecting the widget element on the UI. With the identified element, the tool performs some action, for example clicking on a button. If this results in a change to the visible state of the application, such as updating the date displayed, an assertion may be added to ensure the system responds appropriately.



(a) Main Activity (b) Add a Task Activity

Figure 1. The Simple To Do Application

As an example, consider the Simple To Do application (figure 1) which is an open source application available through F-Droid[3]. The application allows end users to manage their daily tasks and set reminders. There are 10 user tasks associated with various parts of the system functionality. Users have the ability to add, edit, remove and search tasks in the to do list, along with the ability to view and modify certain settings (such as changing the date or time format or creating and restoring backups). For each of these tasks it is important to ensure that the UI responds as expected to user interactions so that they can maintain their to do list accordingly.

[3] See https://f-droid.org/en/packages/apps.jizzu.simpletodo/

Listing 1.1. Dismiss What's New Pop Up

```
1    @Test
2    public void testDismissWhatsNew() {
3       onView(withText("Continue")).perform(click());
4       onView(withId(R.id.coordinator)).check(matches(isCompletelyDisplayed()));
5       onView(withId(R.id.tasksList)).check(matches(isCompletelyDisplayed()));
6    }
```

Listing 1.1 gives an example of an Espresso test script that has been manually written for when the end user of the app wants to dismiss a pop up, in this case the *What's New* popup that opens when an update is installed for the Simple To Do application. This is an important scenario of use to test as the user must be able to dismiss the pop up correctly or it will block them from interacting with the intended functionality of the application. On line 3 the tool locates the continue button (using matcher *withText*) which will dismiss the pop up and performs a *click* interaction. On lines 4-5 we ensure that the correct layout is then displayed to the user (using matcher *withId*). Note that here we omit the rest of the code which is used to initialise the test. Furthermore, in a typical test suite we may have multiple Espresso tests for a single Android application. This allows us to specify different interaction sequences and explore all of the functionality available to the end user.

4. THE TEST GENERATION PROCESS

4.1. Modelling Interaction Sequences

In order to generate tests for Espresso, we must define a process which translates the interaction into test scripts and provides adequate test coverage. To achieve this, we use Interaction Sequence (ISeq) models [13] as the basis for test generation. ISeq models describe user interactions as sequences of actions upon widgets and can be described in text, as finite state automata, or as directed graphs. These models allow us to investigate the behaviour of a system based on user tasks and the interactions required to complete those tasks. They also allow us to define how a user will interact with the different elements of the system and in which order, including exploring sequences of varying lengths for the same task. In the Espresso testing tool tests are created by manually adding interactions via the Recorder, our aim here is to reduce the time required to create test scripts by removing the need for manually writing or recording scripts in Android Studio. Furthermore, should the UI be refactored tests may simply be regenerated from the updated models, further reducing fragility of the tests and the maintenance required to keep the test suite up to date.

ISeq models make use of task ordering to categorise the available behaviours to the user. Using this same ordering, we are able to generate tests for every task that a user may wish to complete, thus covering all possible valid interactions. Consider the Simple To Do application again, in this example we focus on the ability to add new tasks to the to do list. First we start with the direct interaction sequence (the shortest required to complete the task) for adding a simple task without a time and date. This sequence is as follows:

1. Tap AddButton.
2. Tap Title.
3. Tap Keyboard.
4. Tap TickButton.
5. Tap AddTaskButton.
6. Observe Display.

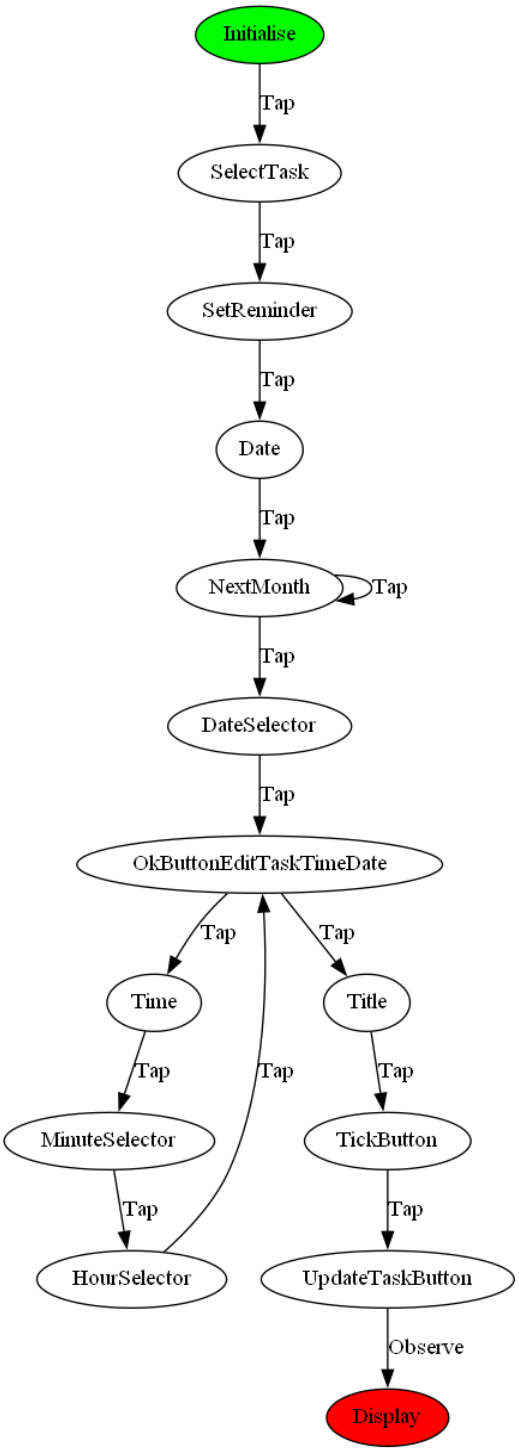


Figure 2. ISeq Model for Edit a Task as a Directed Graph

The user first starts by selecting the add button on the *MainActivity* screen (see figure 1(a)). They then select the *Title* text box and use the *Keyboard* to enter a description for their task (see figure 1(b)). Tapping the *TickButton* confirms the task description and selecting the add task button will tell the system to add this to the list of tasks. The user can then observe the *Display* to ensure the task was added.

While this example specifies a direct sequence, the use of ISeqs also allows us to generate sequences of varying lengths for specific tasks by investigating indirect sequences. Consider the sequence for editing a task:

1. Tap SelectTask.
2. Tap SetReminder.
3. Tap Date.
4. Tap NextMonth 3.
5. Tap DateSelector.
6. Tap OkButtonEditTaskTimeDate.
7. Tap Time.
8. Tap MinuteSelector.
9. Tap HourSelector.
10. Tap OkButtonEditTaskTimeDate.
11. Tap Title.
12. Tap TickButton.
13. Tap UpdateTaskButton.
14. Observe Display.

Figure 2 gives the visual representation of the ISeq model as a directed graph. The green state *Initialise* simply represents the starting state for the sequence while the red state *Display* represents the final state, that is the last widget that should be interacted with. Note the loops on the *Next Month* and *Ok Button Edit Task Time Date* states, these represent interactions that may take place multiple times in a valid interaction sequence. By generating sequences that exploit these loops we may explore indirect sequences for the same task.

Using ISeq models as the basis for test case generation provides adequate coverage of all valid sequences in our test suite. However, there is further work required to investigate "invalid" interactions (interaction sequences that do not complete a task in the specified way) and we discuss this later. Next we define the mapping which allows us to generate Espresso tests in the Java programming language by connecting the ISeq models with an implementation of an Android UI.

4.2. Modelling Sequences and UI Implementation

In the modelling process for the interaction sequences we abstract implementation details of the system in order to focus solely on the interaction and ordering required to complete user tasks. However, in order to use these models as the basis for test case generation, we need a way of connecting the widgets in the interaction sequence with the widgets in the UI of the Android application. To do this we define mappings between the models. For example, in our Simple To Do Application the *MainActivity* mode in Figure 1(a) has a class called *MainActivity* and an identifier of *coordinator*. This allows us to connect the implementation to the model so that we know how to generate the code for the Espresso test script. It is important to make a distinction between different modes in the implementation as while widgets with the same name and behaviours may seem to be the same widget in the model, they may have different identifiers in the implementation.

Within each mode we must create a mapping for each widget in the ISeq model. This mapping contains the name of the widget in the mode, the matcher for the widget (as described in section 3), and the required description. For example, consider line 3 of listing 1.1. In order to generate this line of code for the continue button we need to know three things: the matcher function, in this case *withText*; the description for the matcher, for instance the text identifier *Continue*; and lastly, the action to be taken (*click*). The mapping allows us to define the first two items, the matcher function and the corresponding description, while the ISeq model defines the action. Defining these mappings for each widget will allow us to generate the necessary code for each ISeq.

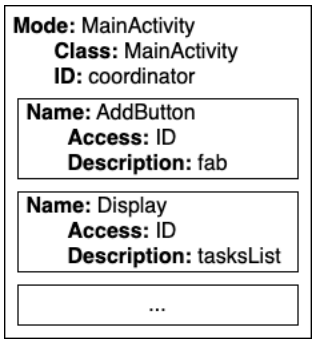


Figure 3. Simple To Do App Mapping

Following on from the Add a Task ISeq model in section 4.1, the relevant mappings for the Main Activity (figure 3) are given. Here we can see that in the *MainActivity* mode the *AddButton* widget can be accessed in the Espresso tool by using a matcher on the implementation's identifier for the widget *fab*. Similarly, we can access the display by looking for the *tasksList* widget in the *coordinator* layout. Using this information alongside the ISeq models we can then generate test scripts that can be imported directly into Android Studio projects for use with the Espresso capture/replay testing tool.

5. THE ESPRESSO GENERATOR

The Espresso generator is the tool we have created that implements the process described in the previous section. Using the models and mapping it generates a test for each task represented by an ISeq model. Sequence generation is done pseudo-randomly, in that the sequence does not need to be direct but must be valid (i.e. end in an accepting state). These tests are saved as Java files which can be imported directly into Android Studio.

Table 1. Supported Espresso Functions

Function	Matcher	Actions
onView	withID	typeText CloseSoftKeyboard
		Click
	withText	Click
	isAssignableFrom	PickerActions.setDate
		PickerActions.setTime
	withContentDescription	Click
onData	withTitle	Click
	withKey	Click

In Table 1 we define the list of currently supported Espresso functions and available actions. Note that not all interaction sequences always require a *click* action. Some actions, such as entering a specific date, are more complex. This requires creating a unique mapping for a date picker view in the Android application. Similarly, typing text with the keyboard and closing the keyboard also requires a distinct mapping as these actions are more complex in the Espresso test scripts. While a limited number of functions are currently supported, we plan to extend the functionality of the Espresso Generator in future work.

We are able to generate assertions that ensure the correct mode is active when a new activity is started or fragment is shown. These assertions can be inserted at different points of the ISeq. This

required us to find a reliable method for creating these assertions regardless of the way the implementation executes these changes. That is, sometimes developers use intents as expected, but this is not always the case. We can check these mode changes by matching the layout identifier for a particular mode (stored in the mapping) and checking that it is displayed. For example, in the Simple To Do Application if we start in the *MainActivity* mode and click the *AddButton* the system should move to the *NewTask* mode. By generating an assertion to check the correct activity is displayed we can confirm the application responds to the interactions as expected.

Some widgets in the UI trigger different parts of underlying functionality of an application. In terms of the Espresso Generator tool, creating assertions for these from the interaction models is complex, as these behaviours may trigger changes which may or may not be visible on the UI. This is not uncommon in capture/replay testing and typically means that tests require human reasoning to determine when these changes occur. While the Generator cannot make these assumptions correctly, as it requires more detailed information, it can generate test prompts which allow a tester to fill in information at a later date. These allow a tester to quickly identify which parts of a test to fill out manually and thus speed up the testing process. In future work we plan to explore this further to increase the automation at these points.

Once all of the test files are generated for each ISeq model the tool exports these as Java files. These files can be imported directly into Android Studio and only need the package names updated in order to be executed as part of the existing test suite. In the scripts "TO DO" tags are added where small package changes are required. This can be done quickly using Android Studio's auto-complete features.

5.1. Generating Tests for the Simple to Do Application

To illustrate the applicability of our technique on a "real world" application we downloaded the source code (version 1.1) for the Simple To Do Application and imported it into Android Studio (version 4.2.2). We used existing ISeq models that were reverse engineered from the implementation for the purposes of this example and imported these into the Espresso Generator along with the mapping. For the purpose of our example we focused on one of the high priority user tasks for this application, adding a task to the to do list. Using the imported ISeq models we can determine what behaviours should be triggered throughout the sequence and how the system should respond.

Using our earlier interaction sequence from section 4.1 for this task, the system begins in the *MainActivity* mode and on step 1 the *AddButton* is tapped. This interaction triggers the add a task activity and the application shifts into this mode. In the add a task activity, step 2 triggers the *Title* widget which responds to two behaviours, setting the task title and updating the text displayed. Step 3 triggers the *Keyboard* widget which allows the user to enter the name of the task. This is followed by step 4 which sets the task title when the user taps the *TickButton*. In step 4 tapping the *AddTaskButton* triggers a change back to the main activity and tells the system to save the task. Lastly, back in the *MainActivity* mode the user observes the display to see that the new task has been added to the to do list.

Using the mapping given earlier in section 4.2 we show how the Espresso Generator will create the code for this sequence. There are three types of identifiers used in this sequence, they are *ID*, *Keyboard* and *KeyboardClose*. The mapping gives a description for each identifier depending on the type. For example, when working with the *ID* identifier, this means we can use the *withId* function to access the widget and perform the action. Similarly, the *Keyboard* and *KeyboardClose* work by using identifiers but instead of performing a click function they perform

the in built Espresso functions of *typeText("Hello World!")* and *closeSoftKeyboard()* respectively. The test generated by the Espresso Generator is as follows:

Listing 1.2. Add a New Task Test

```
1   @Test
2     public void testAddANewTask() {
3       onView(withId(R.id.fab)).perform(click());
4       onView(withId(R.id.container)).check(matches(isCompletelyDisplayed()));
5       onView(withId(R.id.taskTitle)).perform(click());
6       onView(withId(R.id.taskTitle)).perform(typeText("Hello World!"));
7       onView(withId(R.id.taskTitle)).perform(closeSoftKeyboard());
8       onView(withId(R.id.addTaskButton)).perform(click());
9       onView(withId(R.id.coordinator)).check(matches(isCompletelyDisplayed()));
10      //TO DO: add the correct display data for the following assertion
11      onView(withId(R.id.tasksList)).check(matches(/*insert data*/))
12      onView(withId(R.id.tasksList)).check(matches(isCompletelyDisplayed()));
13    }
```

In listing 1.2 a test generated by the Espresso Generator tool for the interaction sequence is given. Here we can see how the different models and mapping allow us to generate different parts of the test. For example, the first step of the sequence is to *Tap* the *AddButton*, in terms of Espresso this translates to clicking the button. Using the mapping we can see that the *AddButton* uses an *ID* matcher with the description *fab*. Therefore the Generator knows it can access the Add button using the *withID* function and passing the Android identifier *R.id.fab*; it also knows that it should perform some action, in this case the *click()* action (as shown on line 3). We know from the models that the system should be in the *NewTask* mode. Therefore, on line 4 we assert that the view is completely displayed. If this assertion fails then we know that the system did not transition to the correct mode.

On line 6 we see a keyboard interaction where the user is expected to type in some text for the title of the task, this requires the tool to enter some text. The tool provides the default text "Hello World!", however, this could be modified by the tester if required. The Espresso Generator knows how to handle this differently as the mapping file specifies a Keyboard access matcher as opposed to an ID. Following on from this, line 7 tells Espresso to close the keyboard associated with the title text box, again, it relies on the ISeq model and mapping to inform the structure of the test.

While the generator does not currently attempt to test the underlying functionality of the system, we would expect a capture/replay tool to at least acknowledge a visible change in the UI in the test. Line 11 acknowledges the result of adding a new task, which is that the new task would be visible on the display. However, we do not know what the output of that behaviour is as it is not given in the models, therefore we use a test prompt to ask the user to update the test with the correct information. At the completion of the test we ensure we have transitioned back to the *MainActivity* mode (line 12).

5.2. Running a Test

The Espresso Generator will create a test for every ISeq model that is given for a particular Android application. This results in a single Java file with multiple tests. In the case of the Simple To Do application we have 10 generated tests for each task of the system, thus ensuring all the functionality of the UI is explored. However, in order to run these tests some minor set up is required by the tester.

We must import the Espresso Generator tests into the Android Tests folder and update the package name. In the case of the Simple To Do application this is the *apps.jizzu.simpletodo* package. Note that this can be done using the built-in autocomplete functionality in Android Studio. Similarly, there may be test prompts that testers will need to fill in before the tests can be run. This can be completed quickly by simply searching for the "TO DO" tag throughout the code. Each of the completed test scripts can be executed like any other test script written for the tool. Espresso will replay the tests and report on any issues that occur, which will allow the tester to debug and update the system accordingly.

6. DISCUSSION

One of the primary benefits of our approach versus traditional capture/replay testing with Espresso is that it does not require a tester to either record tests by hand or write the test scripts. Recording the tests requires significant effort on the part of the tester, as they must use the Espresso Recorder to manually input each interaction sequence one-by-one and generate the test script. Note that the test script generated is quite verbose and difficult to follow. For our test given in listing 1.2 the Espresso Recorder creates a script with 111 lines of code while the Espresso Generator tests the same functionality with only 10 lines of code.

Furthermore, the test script created by the Espresso Recorder is difficult for a human to follow and understand. For example, line 3 of listing 1.2 is given as follows in the Espresso Recorder:

Listing 1.3. Add a New Task Test Line 3 from Recorder

```
ViewInteraction floatingActionButton = onView(
        allOf(withId(R.id.fab),
           childAtPosition(
              allOf(withId(R.id.coordinator),
                 childAtPosition(
                    withId(Android.R.id.content),
                    0)),
                 4),
              isDisplayed()));
floatingActionButton.perform(click());
```

Note that the Recorder takes into account the position of the widget and whether or not the widget is displayed before performing an action. This makes the tests from the recorder more fragile as they must match with all of the conditions given as opposed to only with the unique identifier for the button (i.e. because the identifier is unique the other matchers are unnecessary). If this test were to break, in that Espresso could not run the test, it would be difficult for a human to determine why this matcher failed. The Espresso Generator fixes this issue by simplifying the structure for generated matchers.

In order to write the test scripts the tester needs a clear understanding of the identifiers used in the resources files of the project in order to test different widgets accordingly. This can be a time consuming process and requires the tester to follow a white-box testing approach. However, the tests written by hand do result in fewer lines of code when compared with the Espresso Recorder. The Espresso Generator has been designed to mimic the types of tests that a manual process would produce, however, by using the models it takes a black-box testing approach, in that the mapping specifies the information the tester would need to know in order to write the tests. This reduces the amount of information the tester requires in order to test the UI resulting in time savings.

Similar to the Espresso Recorder, the time consuming nature of writing Espresso tests is reduced by the Generator, as the tester is not required to produce the test scripts manually. However, as stated previously, the complexity of the scripts produced by the Recorder increases the likelihood of fragility in the tests as a direct result of the additional matchers. By simplifying these in the Generator to follow the manual process a tester would take we have reduced the fragility of the generated test scripts. Furthermore, if a change occurs in the system (resulting in an update in the models and/or mapping), we simply need to regenerate the Espresso tests. This reduces the amount of maintenance and upkeep required for testing the UI, resulting in further time savings.
While our model-based testing process provides several benefits for simplifying the capture/replay Espresso testing tool for Android applications, there are also limitations to this method. Firstly, the models must be maintained and updated in order for the tests generated to be accurate. In a model-driven development process these updates would already be taking place, however, if this development process is not being followed then we have increased the maintenance required to update the software and consequently tests.

This increase in development time is a well-known limitation of all modelling processes and not unique to our approach. However, given the benefits that models provide (e.g. verification and validation, producing less error-prone code etc. [14, p.5]), adding models to any development process provides extra benefits, time and cost savings as bugs are usually found earlier, usually resulting in more streamlined development processes.

The Espresso Generator currently only supports a subset of the matchers available for widgets in Android. While our focus has been to implement matchers that enable a wide range of widgets (such as *withId()*) the tool requires further development to explore and add other options. Similarly, the models the tool uses in generation do not provide all the information required to fully generate the tests, resulting in the addition of test prompts. A future line of research will be to investigate how this information can be added appropriately using existing techniques.

For the ISeq models used in our testing process, other tools exist which allow these to be generated easily in an XML format. These files can be used directly in the Espresso Generator. However, the mapping file must be produced manually at this stage. This is part of ongoing work to ensure that the file can be produced quickly.

Currently, the Espresso Generator works on creating valid interaction sequences for each of the user tasks. The disadvantage of this focus is that we do not investigate unexpected behaviour that invalid sequences (those which do not end in an accepting state) would provide. If following a model-driven development process, it is likely that other processes, such as model-checking, are taking place which will identify these types of undesired behaviours. However, it would be useful to also incorporate these in the test suite for robustness and further validation. We plan to investigate this further and ways in which we can control the resulting state space.

7. CONCLUDING REMARKS

In this paper we have presented a model-driven development approach for generating Espresso test scripts for the UI of Android applications using ISeq models. By taking advantage of the benefits that a model-driven process provides we have reduced the fragility of the test scripts and improved the maintainability of the test suite. The Espresso Generator implements our model-driven process and automates the test creation, resulting in further time savings and enhancing the test suite coverage. The generated test scripts may be imported directly into Android Studio allowing scripts to be run directly by the Espresso capture/replay Tool. Should changes occur to the UI of the system, models may be updated and scripts re-generated as necessary, ensuring the

test suite remains up-to-date. By following this model-driven process developers will be able to easily test their Android applications to ensure the UI behaves as expected.

REFERENCES

[1] Meiliana, I. Septian, R. S. Alianto, & Daniel, "Comparison Analysis of Android GUI Testing Frameworks by using an Experimental Study," Procedia Computer Science, vol. 135, pp. 736–748, 2018, the 3rd International Conference on Computer Science and Computational Intelligence (ICCSCI 2018) : Empowering Smart Technology in Digital Era for a Better Life. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050918315011

[2] S. Negara, N. Esfahani, & R. Buse, "Practical Android Test Recording with Espresso Test Recorder," in 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), 2019, pp. 193–202.

[3] S. Feng & C. Chen, "GIFDroid: Automated Replay of Visual Bug Reports for Android apps," in Proceedings of the 44th International Conference on Software Engineering, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1045–1057. [Online]. Available: https://doi.org/10.1145/3510003.3510048

[4] R. Coppola, M. Morisio, & M. Torchiano, "Mobile GUI Testing Fragility: A Study on Open-source Android Applications," IEEE Transactions on Reliability, vol. 68, no. 1, pp. 67–90, 2019.

[5] M. Wynne, A. Hellesoy, & S. Tooke, The Cucumber Book: Behaviour-driven Development for Testers and Developers. Pragmatic Bookshelf, 2017.

[6] J. Campos, C. Fayollas, M. Gon¸calves, C. Martinie, D. Navarre, P. Palanque, & M. Pinto, "A more Intelligent Test Case Generation Approach through Task Models Manipulation," Proc. ACM Hum. Comput. Interact., vol. 1, no. EICS, pp. 9:1–9:20, 2017.

[7] J. Bowen & S. Reeves, "UI-design Driven Model-based Testing," Innovations in Systems and Software Engineering, vol. 9, no. 3, pp. 201–215, 2013. [Online]. Available: http://dx.doi.org/10.1007/s11334-013-0199-6

[8] A. Gimblett & H. Thimbleby, "User Interface Model Discovery: Towards a Generic Approach," in Proceedings of the 2nd ACM SIGCHI Symposium on Engineering Interactive Computing System, EICS 2010, Berlin, Germany, June 19-23, 2010, N. Sukaviriya, J. Vanderdonckt, and M. Harrison, Eds. ACM, 2010, pp. 145–154.

[9] D. Amalfitano, A. Fasolino, P. Tramontana, B. Ta, & A. Memon, "MobiGuitar: Automated Model-based Testing of Mobile apps," IEEE Software, vol. 32, no. 5, pp. 53–59, 2015.

[10] Y. Cao, G. Wu, W. Chen, & J. Wei, "CrawlDroid: Effective Model-based GUI Testing of Android apps," in Proceedings of the 10th Asia-Pacific Symposium on Internetware, ser. Internetware '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3275219.3275238

[11] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, & Z. Su, "Guided, Stochastic Model-based GUI Testing of Android apps," in Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 245–256. [Online]. Available: https://doi.org/10.1145/3106237.3106298

[12] S. Di Martino, A. Fasolino, L. Starace, & P. Tramontana, "Comparing the Effectiveness of Capture and Replay Against Automatic Input Generation for Android Graphical User Interface Testing," Software Testing, Verification and Reliability, vol. 31, no. 3, p. e1754, 2021, e1754 stvr.1754. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1754

[13] J. Turner, J. Bowen, & S. Reeves, "Model-based Testing of Interactive Systems using Interaction Sequences," Proc. ACM Hum.-Comput. Interact., vol. 4, no. EICS, Jun. 2020. [Online]. Available: https://doi.org/10.1145/3397873

[14] R. Oliveira, P. Palanque, B. Weyers, J. Bowen, & A. Dix, State of the Art on Formal Methods for Interactive Systems. Cham: Springer International Publishing, 2017, pp. 3–55.