THE UNIVERSITY OF WAIKATO
Te Whare Wānanga o Waikato
Research Commons

https://researchcommons.waikato.ac.nz/

**Research Commons at the University of Waikato**

## Copyright Statement:

The digital copy of this thesis is protected by the Copyright Act 1994 (New Zealand).

The thesis may be consulted by you, provided you comply with the provisions of the Act and the following conditions of use:

- Any use you make of these documents or images must be for research or private study purposes only, and you may not make them available to any other person.
- Authors control the copyright of their thesis. You will recognise the author's right to be identified as the author of the thesis, and due acknowledgement will be made to the author where appropriate.
- You will obtain the author's permission before publishing any material from the thesis.

# Automated Capture/Replay Test Generation for Android using Espresso

A thesis submitted in partial fulfillment of the requirements for the degree of

**Master of Engineering with endorsement in Software Engineering**

at

**The University of Waikato**

**Supervised by:**

Jessica Turner

By

Fajer Alblooshy

# Abstract

This thesis presents the design of a new tool for automated test generation in Android applications using the Espresso framework. The primary focus is on designing a robust and efficient tool with the use of the Presentation Model (PModels) to simplify the automation of test case design. By incorporating techniques such as interaction sequence modeling and widget-based abstraction, the tool's design aims to address the challenges of testing interactive systems, which are often timeconsuming and complex. The research includes comprehensively analyzing existing testing methodologies, identifying their limitations, and proposing enhancements to improve test coverage and reliability. Through detailed design documentation and conceptual frameworks, the study demonstrates how the tool can generate maintainable test scripts, reducing the effect required for manual testing. The tool's design is evaluated through theoretical case studies, including various Android applications, to illustrate its potential effectiveness in real-world scenarios. This work contributes to the field of software engineering by offering an approach to automated UI testing, emphasizing the importance of design considerations in developing tools that can meet the evolving demands of software quality assurance.

# Acknowledgment

I want to express my sincere thanks to my supervisor, Jessica Turner, of the Computer Science department, for encouraging, guiding, and supporting me to the highest peak and allowing me to prepare this research. I want to acknowledge my indebtedness and deep gratitude to her for guiding and supervising me throughout the course, which shaped the present work.

Secondly, I thank the Engineering and Computer Science faculty for their help and guidance throughout the year in finishing the research within the limited time. Also, I would like to have the opportunity to be a student at the University of Waikato.

Lastly, I would like to express my deepest gratitude to my family for believing in me and their constant encouragement during the research.

# Declaration of Authorship

The work in this thesis has not been previously submitted to meet the requirements for a Master of Engineering with an endorsement in Software Engineering at the University of Waikato. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made.

# Table of Contents

Abstract ..... II
Acknowledgment ..... III
Declaration of Authorship ..... IV
List of Tables ..... VII
List of Figures ..... VIII
List of Abbreviations ..... IX
1. Introduction ..... 1
1.1 Android Studio and Espresso ..... 1
1.2 Purpose ..... 2
1.3 Scope ..... 2
1.4 Methodology ..... 3
1.5 Thesis structure ..... 3
2. Literature Review ..... 4
2.1 Introduction ..... 4
2.2 Methodology ..... 5
2.3 Results ..... 9
2.4 Summary ..... 19
3. Approach ..... 20
3.1 Hypothesis ..... 20
3.2 Research Questions ..... 20
3.3 Research Overview ..... 20
4. Case Studies ..... 23
4.1 Android Application ..... 23
5. Fragility in ETR ..... 35
5.1 Understanding Fragility in ETR ..... 35
5.2 Identifying Fragility ..... 36
5.3 Summary ..... 37
6. Interaction Modelling ..... 38

6.1 Presentation Model - PModel ........ 38
6.2 Presentation Interaction Model (PIM) ........ 40
6.3 Presentation Model Relation (PMR) ........ 41
6.4 Using PModels ........ 42
6.5 Summary ........ 43
7. PMed Tool ........ 44
7.1 Test Case Design ........ 44
7.2 Tool Design ........ 45
8. Discussion ........ 49
8.1 Design versus Implementation ........ 49
8.2 Limitations ........ 50
8.3 Future Work ........ 50
9. Concluding Remarks ........ 51
9.1 Summary of Key Findings ........ 51
9.2 Implications for the Field ........ 52
9.3 Future Work ........ 52
10. References ........ 53

# List of Tables

**Table 1** Search strings categories and keyword ........ 6
**Table 2** Inclusion and Exclusion Criteria........ 7
**Table 3** Methods used to eliminate unrelated research. ........ 8
**Table 4** Description of techniques and tools used in studies ........ 14

# List of Figures

Figure 1 Android calculator app. .......... 26
Figure 2 Main Activity - Add New List.......... 29
Figure 3 Side Menu .......... 29
Figure 4 Pop-up to name the new list .......... 29
Figure 5 Adding a new task .......... 30
Figure 6 Assigning name to the new task .......... 30
Figure 7 Naming then adding the new task .......... 30
Figure 8 The new task is added to the list .......... 30
Figure 9 Task List side menu .......... 32
Figure 10 Mark all tasks feature .......... 32
Figure 11 Unchecking All .......... 32
Figure 12 Pop-up window to rename the list .......... 32
Figure 13 The new renamed list.......... 32
Figure 14 Deleting all tasks from the list .......... 32
Figure 15 The renamed list. .......... 33
Figure 16 Main Activity side menu .......... 33
Figure 17 Sorting Feature - A to Z.......... 33
Figure 18 Sorting Feature - Sort by Done .......... 33
Figure 19 Deleting All Lists.......... 34
Figure 20 Presence of Jackpack Compose in an app. .......... 36
Figure 21 PModel for Calculator app. .......... 40
Figure 22 PModel for To-Do List app. .......... 40
Figure 23 PIM mapping for To-Do list app. .......... 41
Figure 24 Validating the positions of elements for references. .......... 42
Figure 25 Creating PModel using PIMed tool. .......... 45
Figure 26 PMed Interface .......... 46
Figure 27 The UI of Mapping Elements Feature .......... 47
Figure 28 Example of To-Do List Interaction with the PMed tool. .......... 48

# List of Abbreviations

| Abbreviation | Meaning |
|---|---|
| PModel | Presentation Model |
| PIM | Presentation Interaction Model |
| PMR | Presentation Model Relation |
| MBT | Model-Based Testing |
| UI | User Interface |
| OS | Operating System |
| App | Application |

# 1. Introduction

The number of Android users and Android applications either in existence or in the process of development in the current market is widespread, with common applications such as gaming, messaging, shopping [1], social media, streaming, and work-related applications (apps) [2,3]. With the increasing number of apps, the competition to deliver a compelling, secure, and functional app constantly pushes developers to meet higher user expectations. As a result, developers are pressured with the time and resources to conduct app testing, a long and complex process that may lead them to overlook or pay attention to some aspects of the testing phase.

## 1.1 Android Studio and Espresso

Android Studio [1] is a widely used tool for writing and creating Android applications, recognised for its extensive features that streamline the app development process. It offers a unified environment that supports the development of apps for all Android devices, enables real-time editing on emulators and physical devices, and provides comprehensive testing tools and frameworks.

The Espresso Testing Framework, developed by Google, assists developers with writing reliable User Interface (UI) tests. This open-source framework is embedded in Android Studio to test Android applications with a simple IDE (Integrated Development Environment) that can be adjusted to the testing requirements. One of its essential features is allowing developers or testers to synchronize their test actions with the app's UI.

Mobile app testing evaluates the developed mobile applications to verify their functionality across different devices and operating system versions. It also involves examining the nonfunctional aspects such as consistency, usability, performance, and compatibility before releasing it publicly[1]. The Android automated UI test allows us to check that the behavior of the targeted app corresponds with user interactions. Furthermore, it will enable developers to test in both Java and Kotlin without requiring previous app testing skills to create and manage tests to use it.

[1] 1_https://developer.android.com/

Capture/replay testing is a testing approach that executes recorded manual user interactions with the user interface to create automated test scripts based on these interactions. The Capture/replay testing tools record actions such as clicking, tapping, and typing, allowing developers to create UI tests without requiring previous testing skills or writing test code. This approach is often used due to its ease of use and efficiency in generating test scripts from UI recordings.

The Espresso Test Recorder (ETR) is a component of AndroidX Test in the Espresso Testing framework. It enables users and developers to record their interactions with the application, automatically generating identical tests for execution. These tests have two main components: UI interactions and assertions on View elements. UI interactions include actions users perform while interacting with the app, such as tapping or typing, while assertions validate the presence of elements on the screen. Additionally, ETR records the states linked to these interactions, supporting automatic regression testing of the GUI.

## 1.2 Purpose

This research investigates the fragility of generating test cases using model-based techniques and capture/replay testing tools for Android applications. We then design a tool that uses model-based testing techniques to reduce the fragility of testing in Espresso.

## 1.3 Scope

The main focus of this research is designing a tool that will enable the generation of accurate test cases. The PMed tool design and this report are the project's final deliverables. Our expectation of the tool design is to function with any Android application, regardless of its complexity and language, as long as the models used are in PIMed format with .xml file format. In order to consider the tool design successful, test case generation that can be run directly using the Espresso tool will need to be enabled. If these criteria are met, the tool design would be considered successful.

## 1.4 Methodology

The methodology will investigate the approaches to designing a tool that can generate test cases using model-based techniques for Android applications. Our research hypothesis, which reduces fragility in capture/replay testing tools for Android apps, is introduced above using model-based techniques. Our two research questions are stated as follows:

RQ1: What criteria contribute to minimizing the fragility of tests generated using capture/replay testing?

RQ1: What modeling techniques simplify and reduce fragility testing?

## 1.5 Thesis structure

The thesis is structured as follows: The literature Review describes the current research and work on automated test generation using model-based techniques for Espresso. It also identifies gaps and limitations in the existing methods and tools and addresses any challenges and evaluations. Next is the Approach, which introduces the research questions, followed by the model's section, which presents and discusses the Presentation Model and its use with the selected apps to create testing models. Then, the Fragility in ETR section discusses some common fragilities users encounter while using ETR. Following that, we describe the tool design and the research discussion. Finally, the Conclusion will summarize the essential findings and contributions of the research and its significance and relevance to existing research and tools. It also reflects on the research outcomes, limitations, and future work.

# 2. Literature Review

## 2.1 Introduction

In many fields where user interaction is essential to the operation, such as software applications and medical devices, interactive systems enable system testing like functionality, performance, security testing, etc. For these systems to be deployed successfully and to satisfy users, it is essential to guarantee their efficacy, usability, and dependability. Strict testing procedures are necessary to accomplish this in order to detect possible flaws, evaluate user experience, and verify system behavior in varied scenarios. Such a testing approach is automated testing, which has become crucial in systems development and testing, considering its potential to enhance software quality [4].

Testing interactive systems requires knowledge of testing and understanding of the system under test (SUT), and it is considered time-consuming and challenging, especially with larger, more complicated, and sensitive software [5]. These challenges include a lack of testing experience or inexperience in testing tool usage [6]. Software and application testing, like interactive systems, must ensure software quality and user satisfaction. Automated testing is the key to effective and efficient evaluation of system behavior, functionality, and user interactions across different scenarios. Similarly, Model-Based Testing (MBT) is a technique where abstract models of the SUT are used to derive test cases and has been demonstrated to be a practical approach for complex interactive system testing [7], such as testing medical infusion pumps in [15]. This includes techniques such as the interaction sequences and widget-based models as an abstraction and deriving obligations from formal models [8]. By deriving test models from MBT, testers can ensure comprehensive coverage of interactive application functionality and address its reliability.

In this context, a Systematic Literature Review (SLR) was conducted in order to investigate the current interactive systems testing techniques, methods and tools. Additionally, by systematically collecting, evaluating, and synthesizing relevant studies, SLR provides a comprehensive overview of the current state-of-the-art techniques, methodologies, challenges, and future directions in this field. An investigation of methods to enhance the Espresso testing framework for Android applications by implementing formal models will be conducted. This SLR is structured as follows: Section 2 discusses the methodology of the research questions, which includes the search process, inclusion and exclusion criteria, study selection process, and data

collection. It explains the process of collecting and finding related studies for this project by applying specific criteria and requirements during the selection stage. Section 3 discusses the existing techniques and tools used for interactive system testing, such as the capture/replay approach. As well as any challenges in applying these techniques and tools to testing. Section 4 concludes the SLR and introduces the approach that will be taken for this study.

## 2.2 Methodology

This section will discuss finding the required information related to the topic. A systematic literature review (SLR) is the chosen approach for this research, as it is considered the appropriate method for constructing the research. The process of this SLR followed the guidelines [9] and was also inspired by multiple other literature reviews, such as [10-13]. The resources were analyzed for the inclusion and exclusion criteria process and data collection.

### 2.2.1 Research Questions

Application testing, primarily open-source Android application testing studies, is widely available with multiple tools, techniques, and methods. The following research questions are addressed in this study:

**RQ1:** What existing techniques are used for interactive systems testing using a capture/replay approach?

**RQ2:** What are the challenges they have faced using their approach? And why?

RQ1 allows us to collect all the relevant types of interactive testing techniques, possibly applied to systems or applications where capture/replay method(s) were used. As for RQ2, we further investigate their methods and relevance to this research.

### 2.2.2 Search Process

The search was conducted in well-known digital libraries for scientific literature platforms and using manual methods (e.g., snowballing). The platforms used during the search process are Google Scholar, IEEE Xplore, ACM Digital Library, and Science Direct. The advanced search features were used to narrow down the relevant results. The features in each library differ. Thus, the searching techniques were slightly different. For example, using the advanced search in Google

Scholar does not include the option to specify the type of published document (i.e., book, journal article, etc.), while in IEEE, this feature is available. As for the keywords, there are two categories: 'Android Testing' and 'Software Testing'. In each category, the keywords were applied, as shown in Table 1. The three main search keywords from both categories are Android testing, Model-based testing, and interactive system/sequences testing. The second search method is backward snowballing, which was used after applying the inclusion and exclusion criteria.

***Table 1*** *Search strings categories and keyword*

| Search Categories | Key words |
|---|---|
| Android | Android testing, Android application testing, Espresso testing, GUI testing and Espresso recording |
| Software Testing | Model test case, interactive testing, sequence testing, interactive system testing, widget sequences testing, system model testing, interactive system, model testing, widget sequence, task models, model testing, model-based testing, capture-and-replay tools, Android testing tools |

### 2.2.3 Inclusion and Exclusion Criteria

The search outcome from using the previous keywords resulted in a wide range of academic literature. Therefore, inclusion and exclusion criteria were applied to select the most relevant papers. Each study underwent multiple stages, such as full-text accessibility, study classification type, publication language, and keyword relevance (see Table 2). The criteria in the table need to be presented in the specific order of their application. Full-text view accessibility is the most important criterion, as some digital libraries require a payment method for full-text view. The study classifications of interest are academic reports, peer-reviews, journal articles, and experimental studies. The publication language of the studies must be in English, ensuring accessibility and scientific communication on an international level. These publications must not have been published earlier than 2010; newer publications are preferred as they reflect technological

advancements. Including keywords related to software engineering and software testing narrows the findings and ensures better coverage of our research interests.

***Table 2*** *Inclusion and Exclusion Criteria*

| Criteria | Inclusion and exclusion phases |
|---|---|
| C1 | Includes the search keyword in the title, abstract and body. |
| C2 | Full-text view accessibility. |
| C3 | Relevant to software testing. |
| C4 | Published in English. |
| C5 | Date of publication: 2010. |
| C6 | Academic report, peer-review, journal article and experimental studies |

### 2.2.4 Study Selection Process

The most relevant papers were selected after the inclusion and exclusion criteria. The results of search keywords in the digital libraries were significant, and to gather the most appropriate research, a selection strategy was required. The data was categorized into three categories: identification, screening, and inclusion. Each category follows a simple review stage, as shown in Table 3, where each review stage consists of a method and criteria. For instance, identification covers stages 1 and 2, excluding unrelated studies based on search terms and keywords. Screening covers stages 3 and 4, excluding studies published earlier than 2010 or irrelative abstracts. Finally, studies based on the full text should be excluded.

***Table 3*** *Methods used to eliminate unrelated research.*

| | Review Stage | Method | Selecting Criteria | N resources after exclusion |
|---|---|---|---|---|
| Identification | Stage 1 | Identify relevant studies from data sources | keywords | N = 200 out of 650 |
| | Stage 2 | Exclude studies based on title and keywords | Search terms | |
| Screening | Stage 3 | Exclude studies based on abstract | Interactive system AND Model-based testing OR Android Testing | N = 150 out of 200 |
| | Stage 4 | Exclude studies based on publication date | <2010 | |
| Selection | Stage 5 | Exclude studies based on full-text review | Discuss the testing approach related to method(s) testing for code functionality OR testing Android applications propose solutions. AND include future work OR proposing a solution | N = 50 out of 150 |

### 2.2.5 Data Collection

Analyzing each primary study was required to obtain the relevant data and information for data analysis. Also, the primary sources narrowed the search path categories, which helped extract the relevant information from the selected studies. The data collection was categorized into the following:

- Publication year.
- Source and reference.
- Study classification (journal, thesis, book, and studies).
- Data or information related to the search topic and questions.
- Summary Points.

The first two specifications are the publication year and the study classification. Any document not included in C6 from Table 2 and published earlier than 2010 was excluded. Then, studies were eliminated by specifying the keywords that needed to be found in the title. The second elimination was done by applying similar or other keywords to the abstract. Lastly, the exact keywords were searched throughout the document after eliminating all the irrelevant studies from the previous steps. All documents containing the most keywords were considered the most relevant to the study.

## 2.3 Results

This section discusses the answers to the proposed research questions by investigating different testing methods researchers have explored using their testing techniques. Then, we highlight the challenges and limitations they have encountered by defining their method's strengths and weaknesses. Finally, a summary of the research findings is presented.

### 2.3.1 RQ1 – Existing Testing Techniques

Understanding the current interactive systems testing techniques is crucial for understanding the latest practices and methodologies in the field. Analyzing these techniques helps recognize their strengths, weaknesses, and suitability for various contexts. This knowledge assists researchers and practitioners in making informed decisions, optimizing testing processes, and effectively addressing challenges. Table 4 represents multiple testing models, tools, languages, and techniques used for interactive system testing, including model-based, automatic, GUI, and unit testing. The

techniques involve interaction sequences as an abstraction, formal models, interaction sequence models, widget-based testing, and deriving obligations from formal models. The following sections give further explanations of how these methods and techniques were used and how the tools were implemented.

#### *2.3.1.1 Interactive System Testing*

Generating obligations from models of interactive systems to support the development process during programming and testing is used as the foundation for assertions to ensure consistency of behavior between the interaction and functional elements [14]. It establishes a connection between the UI widgets and the widgets' behaviors. The method includes deriving tests, assertions, and obligations from UI models in order to support the development stage.

Similarly, in [15], they applied a formal modeling method to the syringe pump to test its functionality and interactivity. They have also discussed practical problems for checking properties and using storage capacity modeling large systems. Their primary focus is reducing the state space when performing model checking. One method they have discussed is overlooking operations that are unrelated to a specified component of the model.

In order to create a successful testing environment for an interactive system, it is essential to understand the system under test. Interactive systems consist of three components: interactive, functional, and overlap. The interactive component is the system's user interface, consisting of interactive elements and widgets. It also provides access to the system's backend, which is the functional component. The communication link between the two components is the overlap component. In Turner *et al.* [16], the Espresso framework was used to write tests for the functional component and AssertJ for the interactive component. Their testing approach was model-based testing, designed to examine the system's behavior by testing the overlap component using widget interaction sequences. They have used the self-containment property for two purposes: it enables the expansion of interaction sequences. It allows for control of the specific part of the system that requires testing.

Furthermore, they have used the Sequence Simulator tool, developed by Turner *et al.* [17], to generate abstract tests from the interaction sequences automatically. These generated abstract tests ensure valid execution of the sequence by ensuring the availability of the different widgets in

the sequence. The tool is considered critical in the study in ensuring the behavior of the interaction sequences and generating abstract tests. It was developed to modify and manipulate interaction sequence models. To build the tool, they have applied Z and B models for functionality and formal models (PIM and PModel) for the system's interactivity. The Sequence Generator, a supporting tool for Sequence Simulator, is a component of the tool that allows the user to build sequences by selecting steps and displaying them in a separate window. The use of their self-containment property in the tools for model abstraction provides state space expansion when required.

SeqCheck is a model-checking tool that investigates whether certain properties hold for specific models and tasks [18]. As mentioned above, different models are required for different components, like interaction, functional, and overlap components. The model checker also inspects the behavior of triggered interactions and provides feedback. The use of FSA is to model interaction sequences representing the widgets and their interactions, while the ProB tool was used as an extension for SeqCheck to explore the state of the space of the models such as PIM, PMR, and FSA. The utilization of interaction sequence simulation and exploration of the state space of the models have demonstrated their effectiveness in investigating overlap properties.

##### *2.3.1.2 Android Application Testing*

Xu *et al.* have developed Guider (**G**UI Stru**c**ture and V**i**sion co-gui**d**ed T**e**st **R**epair), a tool for automating GUI test scripts for Android applications using intention-based test repair [19]. The development of this approach involved the integration of Guider, a supporting tool, with the Appium framework. This tool effectively utilizes the structural layout and visual characteristics of widgets on apps' GUIs to accurately identify and repair test scripts by comparing two versions of an app. The first version is called the base version and is used as a reference to compare with the second version, which is the updated version. The tool uses the base version of the app to identify widgets that are most likely to change and distinguish them from other widgets. In repairing the test script for different widgets, Guider applies different strategies and compares the structural information of the two versions by matching the relationship between the screens and the widgets. An experiment was conducted using the WeChat app to evaluate the effectiveness and efficiency of Guider. The results showed that Guider was efficient and effective in repairing GUI test scripts for Android apps. Compared with Meter and Wateroid tools, which rely on either visual or

structural information of GUIs for test script repair, Guider enabled a significantly higher number of test actions to run correctly.

In [20], Automatic Input Generation (AIG) tools were compared with students with testing knowledge backgrounds by experimenting with two scenarios. In the first scenario, students did not know the system under test (SUT) and used the Robotium Recorder tool for capture/replay testing. Students' testing results in code coverage were slightly better than those of AIG. Students outperformed the automated tools in the second scenario, especially for long and complex execution scenarios. The AIG tools used were Sapienz, AndroidRipper, and Robo. While AndroidRipper and Robo operate on event-driven and model-based testing techniques, Sapienz operates on a coordinate-based approach. Sapienz can explore random and systematic strategies for test case generation and optimize code coverage. AndroidRipper can explore the dynamically constructed GUI model of the application under test (AUT) using various exploration approaches, such as random and systematic testing. It explores the user's behavior through events triggered by user interaction. Lastly, Robo is a tool that can produce user events by systematic testing strategies, relying on a black-box testing approach while interacting with the applications' UI elements. The comparison between the performance of the tools showed that Sapienz outperformed AndroidRipper and Robo. At the same time, Robo was shown to be the least effective AIG tool.

Dynodroid was designed to generate relevant inputs for Android apps [21]. It views an app as an event-driven program, meaning it interacts with its environment through a sequence of events. These events are processed by the Android framework, which acts as a bridge between the app and the device's hardware and software components. The main principle underlying Dynodroid is the observer-select-execute cycle. In this cycle, Dynodroid first observes the app's current state and determines which events are relevant. It then selects one of these events and executes it, which leads to a new state of the app. The UI events are inputs designed through the app's user interface, such as taps or gestures on the touchscreen. For UI events, the observe-select-execute cycle is relatively straightforward. In the observer stage, Dynodroid analyses the layout of widgets on the current screen and identifies the type of input expected by each widget. To evaluate the tool, an experiment was conducted by comparing the results of Dynodroid, Monkey, and experienced testers. Monkey can generate sequences of random user and system events on the AUT [20]. They have experimented on 50 open-source Android apps, and Dynodroid was found to cover

a significant portion of each app's Java source code. It also outperformed Monkey, a popular fuzzing tool, by requiring fewer events on average to achieve the same coverage.

The Espresso Test Recorder (ETR) utilizes a technique based on the Java Debugger to capture UI interactions and generate robust, well-formed instrumentation tests [22]. The integration into Android Studio allows ETR to use information about the app's code structure to create Espresso tests. The testing methodology employed in ETR is considered a layout-based technique that identifies GUI components involved in recorded events based on explicit layout properties [20]. These properties could include attributes such as coordinates, dimensions, hierarchy, or unique identifiers assigned to UI elements in the layout. The developers tried a few approaches while implementing the tool, such as accessibility service, app instrumentation, modified Android, and low-code Android. Each approach is confronted with challenges discussed in the next section. They have conducted alpha/beta testing, allowing them to understand the scope and limitations of the tool by working closely with users to resolve issues and further improve the tool. Some of their implemented features include pausing and resuming test recording at any point during app exploration to accommodate different testing needs.

BARISTA is a testing technique based on three phases: test case recording, test case encoding, and test case execution. The primary objective of this tool is to assess the functionality of the test through the recording of user interactions, which are prompted by system events that offer a user-friendly interface to establish an oracle. Subsequently, these recorded interactions and oracles are translated into test cases unrelated to any specific device. Lastly, these encoded test cases are executed on multiple devices, and the resulting test outcomes are summarised in a comprehensive report. The testing technique incorporates three key elements: it enables testers to interact with the application by recording their actions. It allows for the specification of expected results using a novel mechanism. The technique automatically translates recorded actions and specified outcomes into a platform-independent test script.

Furthermore, it allows for automated execution of these test scripts across different platforms. To examine the tool's effectiveness, tests were generated from 15 Android applications and compared to those generated from Testdroid Recorder (TR) and Espresso Test Recorder (ETR) to record test cases using Robotium and Espresso, respectively. The evaluation showed that BARISTA can effectively encode user-defined test cases as test scripts with embedded oracles, outperforming TR and ETR.

*Table 4 Description of techniques and tools used in studies.*

| Ref | Testing Models | Tool | Language | Testing Technique |
|---|---|---|---|---|
| [16] | ISeq, PIMs | Espresso, Sequence Simulator, ProB, PModels | Z, Java, Kotlin | Model-Based, Automatic Testing |
| [18] | PModel, PIM, PRM | FSA, ProB Model checker, Prob Automator, Seq Checker | Z, Java | Model-Based, Model-Checking |
| [23] | PModel, PIM, PRM ISeq Models | FSA, FSM | Z | Model-Based |
| [17] | PModel, PIM, PRM, formal models, ISeq models | PIMed, FSA builder, ProB, Espresso, Sequence Simulator | Z, Java | Model-Based, Automatic testing |
| [24] | PModel, Abstract Models | FSM, FSA | N/A | System Testing |
| [14] | Formal models | PIMed, FSA | Z, Java | Model-Based testing |
| [15] | PModel, PIM, PRM | ZooM, ProZ | Z | Model-Based testing, Model-Checking |
| [25] | PModel, PIM, Unit Testing, formal models | UISpec4J | Java, Z | Model-Based testing |
| [26] | Interaction Sequence of Widget | Espresso Test Recorder, MATE, Stoat | Java, Kotlin | Widget-Based testing |
| [19] | GUI Test Script | GUIDER, METER, WATEROID, GUI Ripper | N/A | Intention-Based Test Repair |

| [22] | Interaction Sequence of Widget | Espresso Test Recorder | Java | Record/ Replay |
|---|---|---|---|---|
| [26] | PIM, PModel, PRM, Formal Models | PIMed tool, ZooM, ProB | Z | Model-Based Testing |
| [20] | Branch, Line of Code coverage. | Robotium Recorder, Eclipse, AndroidRipper, Sapienz, Robo, Emma, Firebase platform | Java | Search-Based Testing |
| [28] | GUI testing | Appetizer, Monkey Runner, RERAN | N/A | Record-and-Replay |
| [21] | GUI testing | Dynodroid | Java | Event-driven |
| [29] | GUI testing | BARISTA | N/A | Capture/Replay |

Table 4 shows the type of testing methods each researcher used, the tool they used to test their testing technique, and the testing language. Various techniques and tools have been used in studies related to testing models, including ISeq, PIMs, Espresso, ProB, and Sequence Simulator. These studies have utilized Z, Java, and Kotlin for model-based and automatic testing. Other techniques and tools mentioned in Table 4 include model-checking, FSM, FSA, and widget-based testing, using tools like ZooM, ProZ, and Robotium Recorder.

### 2.3.2 RQ2 – Interactive System Testing Challenges

The challenges and limitations researchers have experienced while conducting their research provide a better understanding of the research context. These challenges reveal unanswered questions or might lead to new paths that require further investigation.

#### *2.3.2.1 Interactive System Testing*

In [24], they have discussed several approaches to exploring interaction sequences as a model, such as removing nondeterminism and minimizing interaction sequences in interactive systems. However, this approach resulted in a loss of information in the models. Another approach they have explored is task-widget-based, resulting in a loss of behavioral information. Using interaction

sequences as models is another approach; however, it presents challenges like the potential for excessively long sequences, which could complicate creating manageable models [24]. With further research and investigation, they could generate interaction sequences using formal models in interactive systems [16-18]. The following section discusses the challenges and limitations encountered while designing, implementing, and testing methods, techniques, and tools.

The purpose of testing an interactive system is to ensure that the output of the designed system behaves as expected. The interaction sequence method was designed to allow the testing of the interactive and functional components separately while testing the overlap component. This required using the self-containment property to constrain the sequence, which offers control over state space by focusing on the tasks that need testing. As mentioned previously, the researchers have used the Sequence Simulator tool to generate interaction sequences (ISeqs), which reduces the manual work and provides control over the size of the state space. ISeqs allowed them to evaluate interactive systems by combining available interactions with the functionality that responds to those interactions. This provides a unique view of the overlap between the interactive and functional components of the system. However, parts of the sequence had to be ignored to adapt the interaction sequences as they needed to shift their focus to the specific testing objectives. This could mean the tool may need more flexibility in multiple testing scenarios with different systems. Another challenge with the simulator tool was automatically generating abstract tests based on the user's assumptions. This may indicate limits to the tool's ability to generate tests effectively, especially in the semiautomated testing process.

According to the available information, SeqCheck has a few limitations. The tool requires manual input of task constraints and the user's running of a checker to identify discrepancies, which can be time-consuming. Furthermore, SeqCheck has yet to be tested on complex systems to prove its effectiveness on different interactive systems.

#### *2.3.2.2 Android Applications Testing*

Throughout the evolution of Android development, many testing tools have emerged to aid developers in ensuring the reliability and functionality of their applications. Among these tools are Espresso Testing Recorder (ETR), Robolectric, and Guider, each offering unique advantages and facing distinct challenges. ETR, a relatively new testing tool developed by Google, stands out for its cross-device compatibility and the production of human-readable test code. However, users of

ETR may encounter difficulties in modifying test scripts and face issues with delayed processing and loss of context when the application's state changes. Meanwhile, Robolectric offers significant speed and ease of use advantages by simulating the Android framework's behavior on the JVM. Yet, it may need help accurately replicating certain Android APIs and complex UI interactions. Similarly, Guider provides a reliable solution for repairing GUI test scripts but faces limitations in testing apps with frequent updates and varying device configurations. Regardless of the tool used, developers are often required to possess a deep understanding of the system architecture and code and may need to extend the capabilities of the tools to address specific scenarios or functionalities.

Dynodroid can effectively generate relevant inputs for unmodified Android apps by viewing them as event-driven programs and monitoring their reactions to events. This approach allows for the interleaving of inputs from both machines and humans, leveraging the strengths of both input generation methods. Dynodroid has demonstrated higher code coverage than manual app testing and the Monkey tool. It successfully identifies multiple bugs in open-source Android apps and the top free apps on Google Play.

BARISTA's record once-run-everywhere testing technique allows recorded tests on one platform to be executed on any other platform without modification or adaptation, streamlining the testing process and enabling testers to reuse their recorded tests across multiple platforms. Additionally, it offers a user-friendly way for testers to create oracles, eliminating the need for specific skills or knowledge. The encoded test cases ensure that tests function correctly regardless of modifications to the user interface, reducing the need for extensive test case modifications or re-creation with each UI change. Moreover, this approach minimizes intrusiveness as it does not require the instrumentation of the AUT.

Despite the advantages of Dynodroid and BARISTA, both approaches encounter challenges that limit their effectiveness in specific scenarios. Dynodroid lacks support for testing apps requiring complex user interactions or custom UI components, and it does not automatically generate test cases based on app specifications. Moreover, it struggles with app-specific permissions and interactions with external devices or APIs. Similarly, BARISTA faces limitations such as the inability to handle complex multi-touch gestures and difficulties with Android apps relying on bitmapped elements. Additionally, both tools grapple with maintaining accuracy in

generated test cases during changes in the application's UI, and they may encounter compatibility issues with other testing frameworks.

In [28], a comparison of three capture/replay tools—Appetizer Toolkit, MonkeyRunner, and RERAN—was conducted to evaluate their effectiveness in testing industrial applications. Notably, RERAN demonstrated coordinate sensitivity, a desirable trait in capture/play tools, and managed to replay most scenarios. However, it struggled with one Facebook scenario, raising uncertainty due to a lack of source code. Additionally, repetitive testing revealed mixed success rates. Appetizer, though more straightforward to use than MonkeyRunner, encountered limitations, failing in specific scenarios on Facebook and WeChat due to tool constraints and unexpected loading times.

Conversely, MonkeyRunner faced challenges recording on multiple devices simultaneously and needed long-press functionality. The results of [20] also indicate that capture/replay may have limitations in testing some scenarios. These findings suggest that a combination of manual exploratory testing and automated testing may be a practical testing approach.

#### *2.3.2.3 Test Fragility*

Based on what has been discussed previously, test fragility refers to the susceptibility of test cases to break or fail when changes or modifications are made to the application under test. Fragile test cases require interventions or updates to accommodate these changes, which can be timeconsuming and resource-intensive. These tests can lead to false positives or false negatives [30], undermining the reliability of testing outcomes. Test fragility can arise from different factors, such as variations in functionality, where changes in the functionality of mobile applications can affect the expected behavior, leading to test failures. Another factor is modification in the GUI. Even small changes, such as the GUI elements' arrangement or appearance, can impact the test cases that rely on the specific elements or their properties [31]. Some other factors could be improved test case design, dependency on external factors, and lack of test maintenance. The majority of the Android automated testing tools cover at least one factor.

Fragile tests can increase maintenance effort, as developers and testers need to constantly update and fix the test cases to accommodate changes in the application. They can also reduce test effectiveness, as they may not reflect the application's current behavior, decreasing its reliability

and overall quality. For instance, a minor UI change in a mobile application can cause previously passing tests to fail, illustrating the fragility of tests in mobile app testing [15].

## 2.4 Summary

Researchers have great potential to further investigate the domains of capture /replay testing tools, model derivation, and interaction sequences to enhance the effectiveness of software testing methodologies. It is important to note that the examination of similar task techniques underlines the significance of capturing/replaying interaction sequences, and the analysis of abstraction with interaction sequences highlights the possibility of deriving models from captured interactions to augment testing environments. Additionally, exploring various testability transformations emphasizes the necessity of improving the dependability of synthesized tests, thus suggesting a focus on effectively capturing and utilizing interaction sequences. As researchers continue to explore these avenues, their objective is to refine testing strategies, enhance the capabilities of testing tools, and ultimately advance the field of software testing.

The project aims to improve the effectiveness of testing Android applications by combining model-based testing with a capture/replay Android testing tool to generate test cases and mitigate test fragility. Using testing models, particularly PModels, and deriving test cases from them could reduce test fragility. By abstracting the application's behavior into PModels, any changes or updates can be reflected and edited within the PModels themselves. Since the test cases are derived from these models, they automatically inherit the updates, ensuring they remain aligned with the evolving application. This approach could minimize the risk of test fragility caused by changes in the application's UI, logic, or environment. Instead of directly modifying individual test cases, testers can focus on updating the PModels, leading to more efficient maintenance and greater resilience against test fragility.

# 3. Approach

This section will discuss the approach to designing a tool using model-based testing techniques to mitigate fragility in capture/replay tools for Android applications. We introduce the hypothesis and associated research questions, followed by the research overview outlining the design requirements.

## 3.1 Hypothesis

Model-based testing techniques can reduce fragility in Android application capture/replay testing tools.

## 3.2 Research Questions

RQ1: What criteria contribute to minimizing the fragility of tests generated using capture/replay testing?

RQ2: What modeling techniques simplify and reduce the fragility of testing?

## 3.3 Research Overview

This research investigates the fragility of the test scripts generated by the capture/replay testing tool to design Presentation Model (PMed Tool) software that generates automated test cases. The proposed approach involves using modeling concepts and test case generation. A case study will be conducted to support this approach, using the Espresso testing platform and Presentation Models to explain the tool's design for generating test scripts. Designing such a tool requires establishing criteria to meet a software design standard.

### 3.3.1 Design Requirements

The following briefly explains the anticipated criteria contributing to the design process to determine successfully generated tests in the capture/replay testing tool.

*Coverage*

Generated tests should cover a substantial part of the application's functionality, including the range of testing features and how thoroughly each feature is tested. A higher coverage means a more comprehensive test suite and minimizes the probability of undetected defects. Coverage contributes to the efficacy of the testing process by ensuring that the tests include a sufficient

portion of the application’s functionality and codebase. The test coverage metrics this project focuses on are functional coverage, which measures all the functional requirements and features tested to behave as intended.

*Accuracy*

Generated tests must precisely mirror the user interactions recorded during the capture phase to ensure the replayed tests accurately replicate the application's behavior under test. Accurate tests ensure the application’s functionality is valid, ensuring its intended behavior. It also helps detect defects in the app’s behavior by comparing its actual behavior with the expected behavior defined in the test, where inaccuracies or deviations can be identified. In this case, an accurate test means the accuracy and success of replaying the recorded test. The more precise the replayed test with the recorded, the higher the chances that the test scripts will not break. The accuracy of replayed test scripts using PModel is essential, ensuring they precisely mirror the user interactions recorded during the capture phase. Furthermore, it will help ensure the application’s functionality is valid and facilitate the detection of defects. Additionally, the collected information will be used to generate PModels illustrating the behavior of the UIs.

*Robustness*

Generated tests must be robust enough to endure modifications made to the user interface or underlying application implementation. This involves smoothly managing dynamic components, changes in data inputs, and unexpected fault situations without leading to test failures or false positives. Robust tests are resilient to changes in the application environment or underlying technologies. Furthermore, it can detect various defects and anomalies in the application under test. Also, it requires less frequent updates and maintenance than fragile tests, which will help create a stable and resilient testing infrastructure. Robust tests can provide broader coverage of the application's functionality and behavior.

Two important measures have been considered to assess the robustness of the generated tests. First, a key indicator is the number of test failures or false positives during test executions. A reduction in this frequency would indicate that robustness has improved. Second, the tests must be flexible enough to accommodate modifications to the user interface or the underlying application implementation. This can be measured by counting the number of tests that need to be

updated because of changes; the lower the number of updates, the more robust the system is. By monitoring these indicators, we aim to ensure that our testing structure remains stable and resilient, capable of change and modification.

### 3.3.2 Modelling

Modeling provides a structured approach to understanding and representing the UI, improving the testing procedure's reliability. By defining UI components and their interactions precisely, modeling helps create robust tests less susceptible to failures caused by changes in the application's UI or underlying implementation. Eventually, this contributes to a more stable and effective testing process, reducing the probability of test failure and false positives caused by UI inconsistencies or unexpected behavior. Thus, modeling plays a crucial role in addressing the fragility of testing and ensuring the reliability of capture/replay testing tools.

# 4. Case Studies

The selection of proper applications is critical to ensuring the usefulness and relevance of the testing process. In this section, we have selected Android applications for testing purposes and introduced two case studies: a calculator and a to-do list. Each application undergoes a comprehensive evaluation process based on predefined criteria to determine its suitability for testing. We aim to simplify the testing process by following specific selection criteria, optimizing resource utilization, and ensuring alignment with our testing objectives.

## 4.1 Android Application

This section discusses the application selection criteria and introduces the selected Android apps. The chosen applications include a calculator and a to-do list; for each, an overview, reasons for selection, and screenshots will be provided.

### 4.1.1 Application Selection Criteria

Selecting objectives that align with our aims ensures that we address challenges related to the fragility of capture/replay testing tools, resulting in meaningful outcomes. Establishing criteria for application selections prevents time, waste, and resources spent testing irrelevant applications, focusing on ensuring the selected apps are suitable to be tested. The following explains each criterion for selecting applications:

**Complexity:** using different app complexity provides insights into the testing challenges many encounter during the testing process. A more complex app tends to have more features, functionalities, and interactions. The application time and resource complexity during testing may require more code modifications. Furthermore, the variety of app complexities can help evaluate different aspects of their behaviors.

**Dynamic Android Widgets:** Dynamic widgets enable the creation of a more interactive UI by displaying real-time responses to user interactions. Testing dynamic elements could improve the process of verifying the app's behavior with the expected user interactions and identifying any related issues in their behavior, ensuring the app's better performance.

**Functionality and performance:** The application should perform as designed without severe bugs or issues that require fixing. Performance represents responsiveness, speed, and stability under

different conditions. Functionality requires the app to provide the necessary functionalities to complete the basic tasks successfully.

**Latest Version:** The app should be relevantly new, meaning it does not require excessive editing and code updating issues before testing. Minimum code modifications reduce significant time and resources and the need for less additional development involvement.

**Source code accessibility:** The app's source code is crucial for understanding its structure, ensuring better test coverage, and allowing code modification. It also eases test investigation and allows for effective debugging.

**Supported version:** The app's code must be compatible with the phone's operating system version to ensure accurate and reliable testing results.

### 4.1.2 Selected Applications

We introduce two distinct Android apps chosen for testing: a calculator app and a to-do list app. A brief overview of each application's functionality and interface design is given. Furthermore, we provide a detailed justification for the selection of each application, highlighting its relevance to the predefined selection criteria interactions and its potential to facilitate comprehensive testing. Additionally, screenshots illustrating interactions with the app offer a visual representation of their functionalities, helping understand their testing implications.

#### *4.1.2.1 The Calculator App*

*App Overview*

The calculator is a minimalist application with basic arithmetic operations and a simple user interface design consisting of buttons and a text view. The calculator is a static app with fixed content, meaning it does not change based on user interactions. This is an excellent example to use when starting the testing phase. This calculator cannot perform scientific functions or support graphical capabilities.

*Justification for selection*

**Complexity:** The calculator app is selected because it consists of a single state, which simplifies the testing process and lays a solid foundation during the tool design. Because of its simple

functionality has potential time-saving benefits during the design phase in a controlled environment, which can optimize and accelerate the overall design timeline. The calculator's simple functionality helps us stay focused on testing without getting distracted by more complex features.

**Standard Android widgets:** The calculator app has been built using standard Android widgets, consisting of text views and buttons. These widgets are familiar to users, ensuring natural interaction, and they offer a robust foundation for testing and debugging. Starting with an app consisting of standard components allows for more efficient coding and testing during the design process.

**Functionality and Performance:** The calculator provides basic arithmetic operations such as addition, subtraction, multiplication, and division. Its functionality and performance are expected in terms of speed and calculations. However, the calculator does not perform complicated or advanced functions, such as percentages and trigonometry. Furthermore, the app does not contain history, where the previous results would be stored and visible to the users. In terms of performance, the app is fast and accurate when performing simple arithmetic calculations.

**Code Modification:** The application required minimal modifications and updates, which were relatively inexpensive. The only modification needed was a change to Gradle, which automatically imports the code lines for the app to run.

**Source code accessibility:** The application's source code is available on GitHub 2, which allows us to understand its structure and functionality better, which will help us understand the outcomes of our tests. It also allows us to locate and fix any potential errors and bugs accurately, improving the app's execution before and during testing. Furthermore, having the source code will help us ensure the app's functionality and usability are reliable and robust enough for testing.

**Supported version:** Mobile devices differ in size, resolution, and operating system. Therefore, supporting at least one mobile version would ensure compatibility with the device's operating system and resolution, ensuring better code execution. Our device in this research is a Samsung S22, version 6.0.

*Interacting with the app*

Using this calculator app is similar to other calculator applications. Users tap on the number buttons and the mathematical operations they require. Upon completion, pressing the equals button will display the results of the operations. In the screenshot below, a calculation of 40 multiplied by 20 equals 800 (40 * 20 = 800). Notably, this application does not display the operation on the screen; similar to the middle image in Figure 1, it only shows the resulting number without the additional operation.

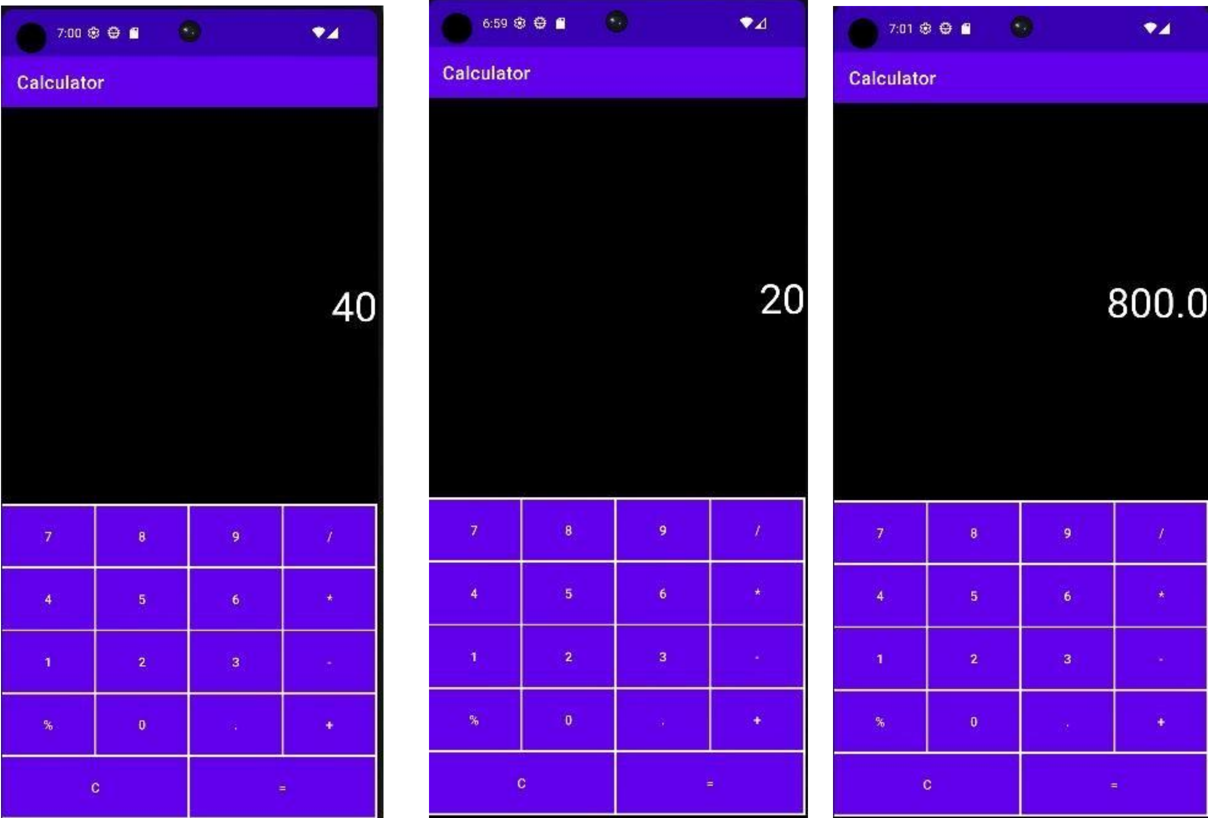


*Figure 1 Android calculator app.*

*4.1.2.2 The To-do List App*

*App Overview*

The do-list app offers users a straightforward platform for creating, managing, and organizing tasks and lists. Its straightforward user interface design allows users to easily interact with tasks and prioritize them according to their preferences. Like complex applications, the simple-to-do list is a dynamic application consisting of multiple states that change based on user interactions with the app.

*Justification for selection*

**Complexity:** The simple-to-do list app was chosen as a more complex user interface with multiple interactive states and a more comprehensive range of features. Using a complex app for testing allows for different testing scenarios for features and functionalities. These scenarios can mimic real-world user interaction in a realistic and controlled environment to test the application's performance. The app's complexity can also help identify any possible errors and bugs in our tool design and testing. A complex app presents a challenge for testing the tool's performance in generating tests that simulate real user interaction, especially in handling complicated and various outcomes.

**Dynamic Android widgets:** Testing apps with dynamic widgets is important because they are changeable and dependable for user interactions with the app's interface. Unlike static apps like the calculator, the simple To-Do app contains a variety of widgets that differ in purposes and functionalities. Testing with dynamic widgets can introduce new errors and potentially affect the tool's performance. Because of the differences in widget functionalities, these bugs may not be detected during static application testing. This will help understand and improve the tool's overall performance.

**Functionality and Performance:** The app's functionalities are not as simple as the mathematical operations since it consists of multiple elements such as deleting, creating, managing, and organizing lists and tasks. This means the user's performance for such an app will consist of diverse outcomes offering a wide range of features. Many currently available applications that users interact with have complex features and functionalities, and app developments are only getting more creative with customized features and interfaces. The purpose of designing and developing

the test generation tool is to be able to generate tests for such applications. Therefore, the simple to-do list is chosen as a step into testing different and more complicated functionalities. As for performance, the app runs at average speed with no delays while creating or interacting with its features. The UI is easy to use, with fast transitions between states.

**Code Modification:** The app's code did not consume time to apply the necessary updates and modifications. Some of the updates needed were the SDK and Gradle versions; no other code modification was required to run the app successfully.

**Source Code Accessibility: The application's source code is available on GitHub 2, which helps us better understand its** structure, functionality, and testing results. It also allows us to identify and address potential faults and errors, enhancing the application's performance before and during testing. Furthermore, having the source code will help us verify the app's robustness for testing.

**Supported version:** Similar device conditions are considered with the Calculator app applied to the To-Do List app.

*Interacting with the app*

**Adding a new list**: The user can create a new list by clicking the '+' button depicted in Figure 2 or by selecting the tab from the side menu shown in Figure 3. A pop-up window will appear for the user to name the new list (Figure 4). Once the user successfully creates a new list, it will be added to the 'Main Activity' list.

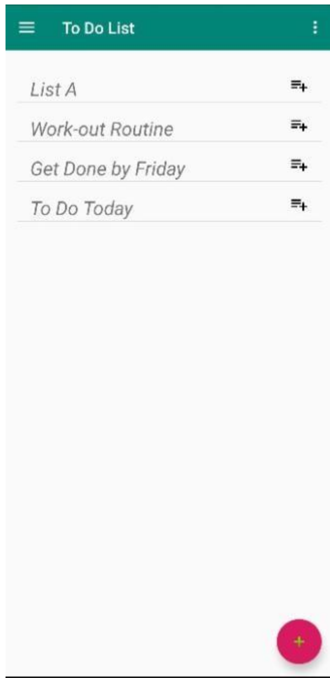


Figure 2 Main Activity - Add New List

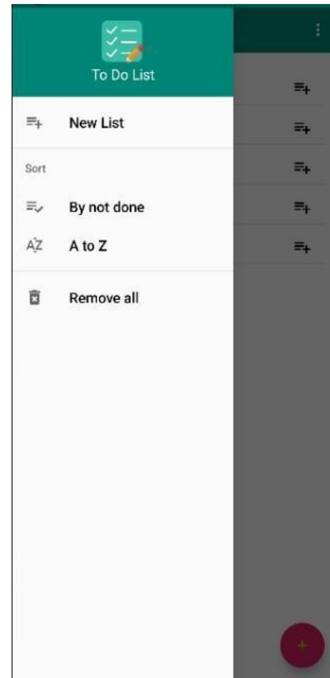


Figure 3 Side Menu

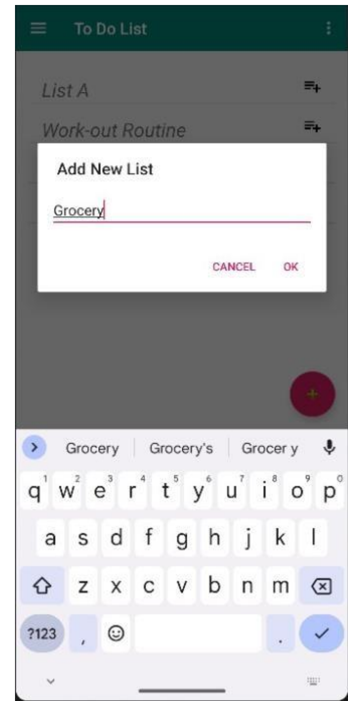


Figure 4 Pop-up to name the new list

**Creating/ adding a new task:** By clicking on a list, a new state will show the content of the list. If the list is new, then it will be empty. But if the chosen list existed and was used before, it will show all the created tasks. To create a new task, the user can click on the '+' button, as shown in Figure 5, and assign a name to the task, as indicated in Figure 6. To add a new task to the list after naming it, the user would click on the green '+' button next to the task's name. Figures 7 and 8 show the new functions being added to the list.

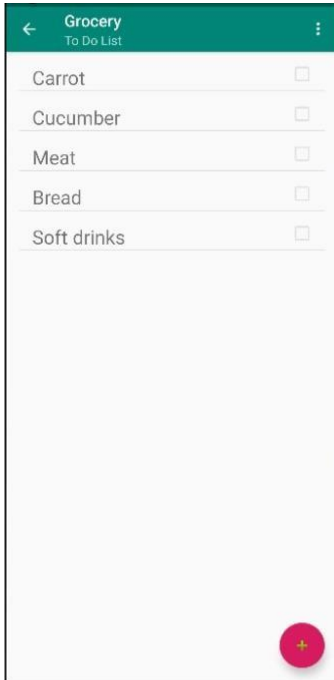


*Figure 5 Adding a new task*

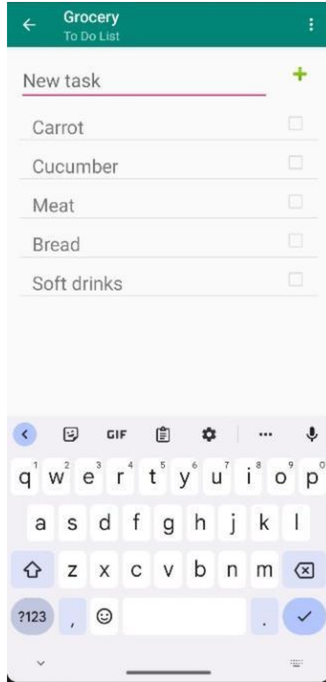


*Figure 6 Assigning name to the new task*

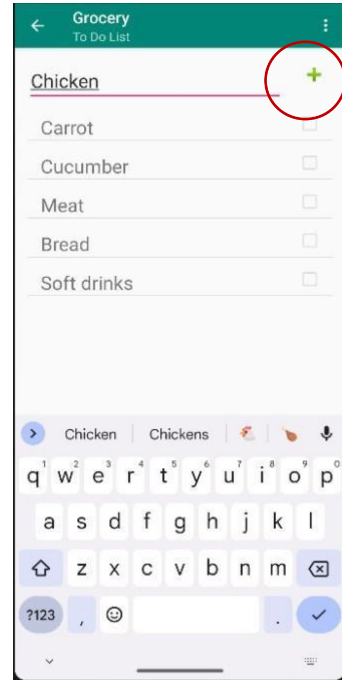


*Figure 7 Naming then adding the new task*

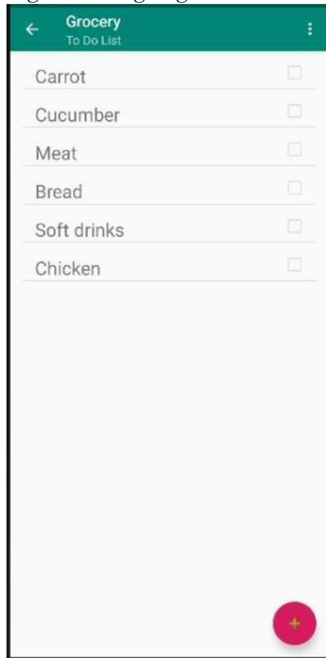


*Figure 8 The new task is added to the list.*

**Side Menu for task list:** Users can execute additional features for the tasks using the side menu in the top right corner, similar to the layout shown in Figure 9. This includes functionalities such as ‘Mark All’ - marking all tasks simultaneously (Figure 10), ‘Uncheck All’ - unchecking all tasks

in the list simultaneously (Figure 11). Also, 'Rename the List' – changes the list name with a popup window to rename Figures 12 and 13. Lastly, 'Clear List' - deleting all tasks within the list (Figure 14). Of course, the user can check some of the boxes manually without the need to check them all at once. It is important to note that users can manually select sure checkboxes without checking all at once, as illustrated in Figure 15.

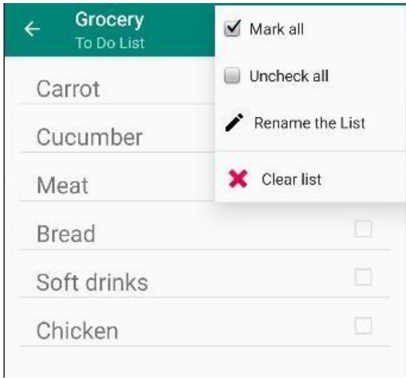


*Figure 9 Task List side menu*

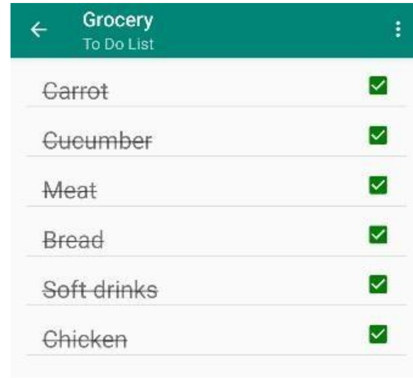


*Figure 10 Mark all tasks feature*

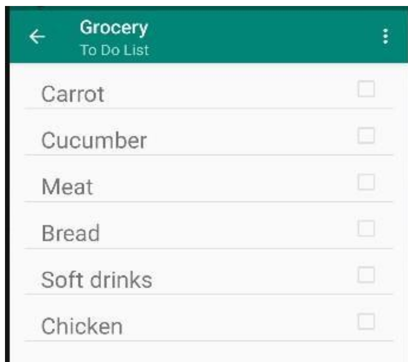


*Figure 11 Unchecking All*

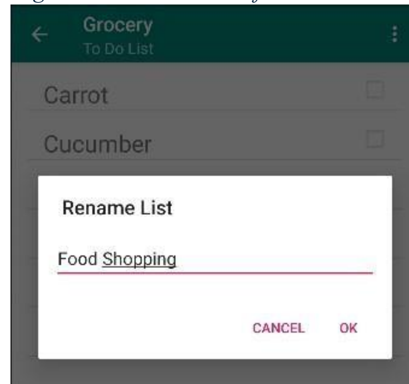


*Figure 12 Pop-up window to rename the list*

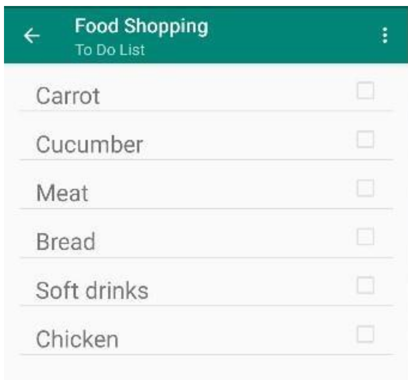


*Figure 13 The newly renamed list*

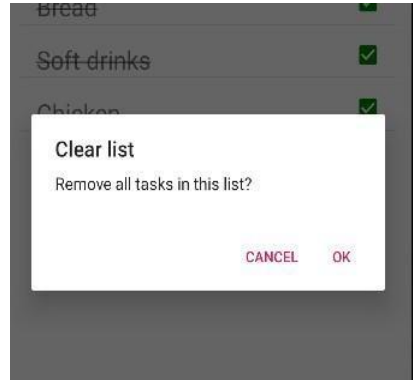


*Figure 14 Deleting all tasks from the list Figure*

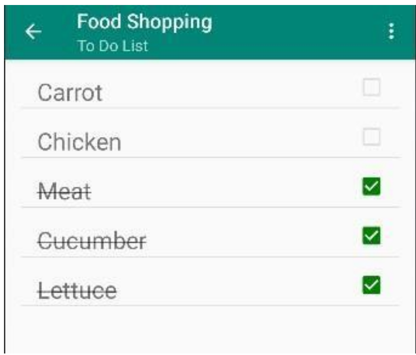


*Figure 15 The renamed list.*

**<u>Side Menu</u> for List:** The side menu options available in the Main Activity are primarily focused on managing lists, as depicted in Figure 16. This initial feature relates to sorting the lists in two ways: 'By A to Z,' which organizes the lists alphabetically (Figure 17), or 'By Not Done,' which arranges the lists by displaying incomplete lists at the top (Figure 18). Lastly, the 'Delete All Lists' functionality removes all lists within the Main Activity, as shown in Figure 19.

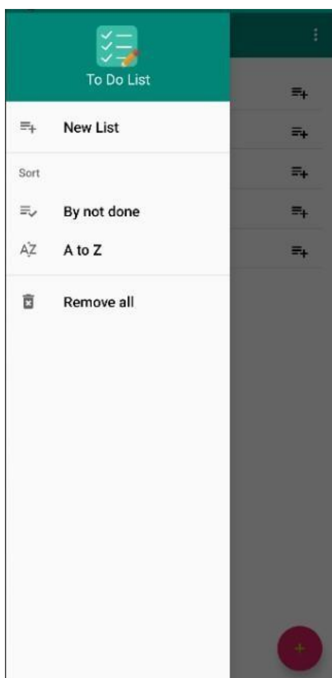


*Figure 16 Main Activity side menu*

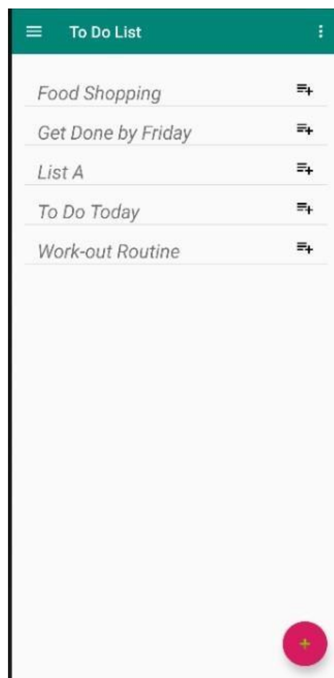


*Figure 17 Sorting Feature - A to Z*

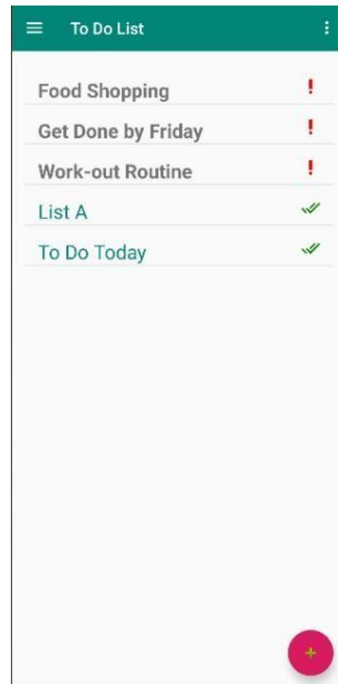


*Figure 18 Sorting Feature - Sort by Done*

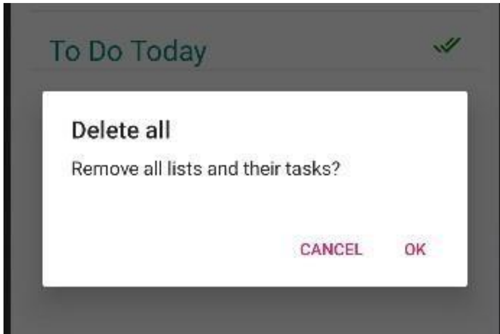

*Figure 19 Deleting All Lists*

# 5. Fragility in ETR

Espresso Test Recorder is a valuable tool for generating automated test scripts for Android applications, offering significant advantages in ease of use and rapid script generation. However, one of the critical issues testers face when using it is the fragility of the generated test scripts. Fragility refers to the susceptibility of the test scripts to break or become outdated because of changes in the application under test or its environment. In this section, we will introduce the common issues users experience while using ETR, but we will first start by understanding its fragility.

## 5.1 Understanding Fragility in ETR

Fragility in ETR results from various reasons fundamental to the process of script generation and the characteristics of the generated scripts:

### 5.1.1 User Interface Script Generation

ETR generates test scripts based on user interactions with the application’s UI elements. This UIcentric approach leads to scripts linked to the application’s UI structure. Any changes to the UI, such as layout modifications or element reordering, can result in failures in the generated scripts.

### 5.1.2 Fragility of Recorded Interaction

The interactions recorded by the ETR, such as clicking a button or inputting a text in textField, are particular and can affect the test's success. For example, if the label of a button changes or a new state is introduced, the recorded interactions would not be effective.

### 5.1.3 Lack of Elimination

ETR does not provide an instrument that helps abstract the test logic from the UI detail, resulting in a script containing complex code line references and sequence actions of UI elements.

Thus, scripts are susceptible to breakage when the UI changes.

### 5.1.4 Error Handling Limitation

The generated scripts often lack robust error-handling mechanisms. When unexpected conditions occur during test execution, such as a delay reaction or UI timeout, the scripts will fail suddenly.

## 5.2 Identifying Fragility

This section discusses the common issues users encounter when using Espresso Test Recorder (ETR), highlighting the fragility of the tool in practical scenarios. By examining the problems, we aim to shed light on the challenges and limitations of ETR in creating reliable and maintainable test scripts.

### 5.2.1 Slow Recording

ETR is known for its slow synchronization with user interactions while recording, whether the app is simple or dynamic. Due to this, users had experienced a hard time interacting with the app while recording, especially with dynamic apps, which most often require multiple steps to test each feature. Such circumstances can result in significant disruptions in accurately capturing user actions. Despite more superficial apps potentially showing quicker responses, recording can still perceived as sluggish. This delay can potentially hinder test generation efficiency, making it challenging to record precise interactions. Furthermore, users are most likely to spend longer on complex applications with extended steps.

### 5.2.2 Presence of Jetpack Compose

The presence of Jetpack Compose in an application automatically removes the ability to use ETR for testing, even if the user intends to record a test against a non-compose activity. This limitation is particularly problematic because it has not been well-documented, leaving users without guidance on how to proceed. As a result, many up-to-date Android applications that utilize Jetpack Compose cannot leverage ETR, significantly reducing its advantage for some modern Android projects. Figure 22 shows a pop-up window for an application that was eliminated due to failure using ETR.

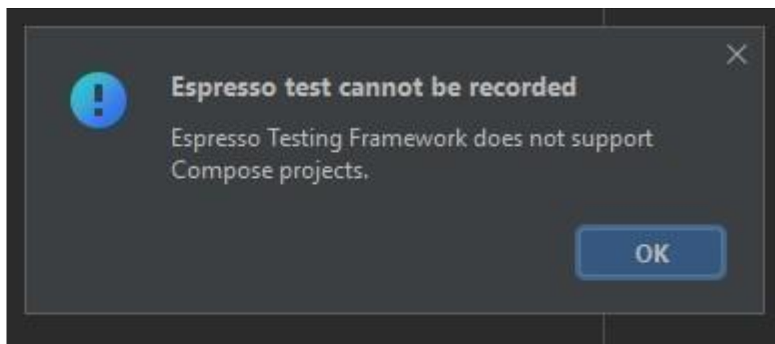


*Figure 20 Presence of Jackpack Compose in an app.*

### 5.2.3 Code Duplication

ETR is most effective for generating simple tests; developers typically create helper classes to streamline app interactions when writing extensive test suites. Since ETR generates pure Espresso code, it can lead to a lot of code duplication across different tests. This redundancy makes it less practical for large-scale test recordings. While ETR can generate specific test statements, relying on it to create entire test scripts can result in challenges in maintaining them.

### 5.2.4 Recording with ETR

One of the disadvantages of using ETR is that it always starts recordings from the app's initial state, which poses a challenge for developers. Developers must repeatedly navigate from the app's initial state to the desired state for each test case, resulting in longer test scripts. It restricts the ability to test scenarios that require the app to be in a specific state. This process is time-consuming and inefficient, slowing down the overall testing process and increasing the likelihood of errors in test scripts.

## 5.3 Summary

In this section, we have explored the fragility issues in the ETR tool, highlighting a few fundamental limitations that affect the reliability of automated generated test scripts. These limitations restrain users from generating comprehensive and easy-to-maintain test scripts, underlining the need for a more robust testing solution. The following section will introduce a model-based technique used on our design testing tool and discuss how PModels can underlie these fragilities.

# 6. Interaction Modelling

The use of formal models in software testing has proven helpful in offering potential advantages in test automation and coverage. Various methods and tools have been proposed or developed to apply formal models in different aspects, including task analysis, interface design, and functional behavior specifications. Presentation Models can be effective in terms of simplifying the flexibility of the Android testing system and the behavior of the interface components. In this section, we explore the concept of PModels, discussing their impact on model-based testing strategies to minimize the fragility of capture/replay tools for Android applications. We examine the structure of PModels, defining their role in capturing interaction and system behaviors within an application's interface. Furthermore, we explain the reasoning behind applying model-based testing strategies that leverage PModels, highlighting their advantage in aligning test cases with the expected functionality derived from the models.

## 6.1 Presentation Model - PModel

The Presentation Model (PModel) is the foundation of the interaction modeling software design as it captures the structural and behavioral facets of the UI, laying a foundation for systematic test generation. Each interactive element, or widget, within the UI, is detailed, covering attributes such as the widget's name, category, and associated behavior. Behaviors in the PModel are categorized into Interaction Behaviour (I-behaviour) and System Behaviour (S-behaviour). The Ibehaviours are user-initiated actions that trigger state transitions within the app; examples include button clicks, text inputs, and swipe gestures. As for the S-behaviours, they are the system-initiated responses to interaction behaviors. They include data retrieval, screen navigation, and background processing.

Each widget has two categories: Action Control and Responder; these categories define the hierarchical structure and establish different roles in software functionality. Action Control is responsible for generating events such as button clicks or checking boxes, which result in a change of event. Responder is primarily associated with display as it responds to these events by displaying the changes. For example, clicking on the 'New List' tap will trigger related events, such as naming the list and confirming the creation of a new one, then displaying the list on the Main Activity screen. The tap is categorized as an action control, and the list is shown as a responder.

Figure 22 illustrates the PModel for the To-Do List app, showing the interactive elements and their interactions. For example, in the PModel for 'Edit List,' we can see widgets associated with either I-behaviour (e.g., RenameList), S-behaviour (e.g., DisplayTask), or both behaviors (e.g., MarkAll, ClearList). I-behaviour involves clicking action, while S-behaviour involves input credentials and navigation to the screen upon successful authentication.

### 6.1.1 Comparison of PModel

Figures 21 and 22 illustrate two distinct PModels for different UI components. These models highlight how variations in UI design and interaction behaviors can lead to other testing scenarios.

Figure 21 depicts a PModel for a calculator application. The primary interactive elements are the numbers and mathematical sign buttons. The associated behaviors are straightforward, involving fundamental interactions and a validation process. Testing this model primarily focuses on verifying the correctness of the mathematical operations and ensuring that the buttons are valid and work as expected.

In contrast, Figure 22 presents a more complex PModel for a multistep process. This model includes additional interactive elements, such as adding a new list or removing items from the list, and input actions, like naming a list or a task. The behavior associated with these elements is more complicated, involving multiple state transitions. Validating this model requires a more comprehensive approach, including addressing various interaction paths and validating multiple UI components.

The differences between these models underline the different levels of complexity in UI design and their impact on testing strategies. Testing the Calculator app model will primarily focus on testing essential input and output handling and the correct operation results. These tests are straightforward and involve direct user action and immediate system responses. On the other hand, the To-Do List app will require comprehensive testing to cover multiple interactions and scenarios. These tests must describe user inputs, state transitions, and relationships between numerous UI components. Such complexity requires test coverage. By understanding the distinct features of these models, we can modify our testing strategies and cases to address the specific challenges posed by each model.

```
MainActivity is

PModel        MainActivity
WidgetName    button1 button2 button3 button4 button5 button6 button7 button8 button9 button0 Minus Multiply Add Divide Percentage Period Delete Equal
Category      ActionControl
Behaviour     I_MainActivity S_button1 S_button2 S_button3 S_button4 S_button5 S_button6 S_button7 S_button8 S_button9 S_button0 S_Minus S_Multiply S_Add
S_Divide S_Percentage S_Period S_Delete S_Equal

MainActivity is
(button1, ActionControl, (I_MainActivity, S_button1))
(button2, ActionControl, (I_MainActivity, S_button2))
(button3, ActionControl, (I_MainActivity, S_button3))
(button4, ActionControl, (I_MainActivity, S_button4))
(button5, ActionControl, (I_MainActivity, S_button5))
(button6, ActionControl, (I_MainActivity, S_button6))
(button7, ActionControl, (I_MainActivity, S_button7))
(button8, ActionControl, (I_MainActivity, S_button8))
(button9, ActionControl, (I_MainActivity, S_button9))
(button0, ActionControl, (I_MainActivity, S_button0))
(Minus, ActionControl, (I_MainActivity, S_Minus))
(Multiply, ActionControl, (I_MainActivity, S_Multiply))
(Add, ActionControl, (I_MainActivity, S_Add))
(Divide, ActionControl, (I_MainActivity, S_Divide))
(Percentage, ActionControl, (I_MainActivity, S_Percentage))
(Period, ActionControl, (I_MainActivity, S_Period))
(Delete, ActionControl, (I_MainActivity, S_Delete))
(Equal, ActionControl, (I_MainActivity, S_Equal))
```

*Figure 21 PModel for Calculator app.*

```
ToDoList is SortingList : MainActivity : EditList : AddNewList : AddTask : RenameList

PModel        SortingList MainActivity EditList AddNewList AddTask RenameList ToDoList
WidgetName    AtoZ NotDone DisplaySort RemoveAll AddList SortingList DsiplayLists MarkAll UncheckAll DisplayTasks ClearList RenameList AddTask Back OKButton
 CancelButton
Category      ActionControl Resnponder Responder ActionContrlol
Behaviour     I_MainActivity S_SortAlphabaticalList S_SortUnfinishedList S_DisplaySortedList S_DeleteLists I_AddNewList I_SortMenu S_DisplayLists I_EditList
S_MarkAllBoxes S_UncheckAllTasks S_DisplayTasks S_DeleteAllTasks I_Rename I_AddTask S_AddList S_AddNewTask S_Rename

ToDoList is   SortingList : MainActivity : EditList : AddNewList : AddTask : RenameList
SortingList is
(AtoZ, ActionControl, (I_MainActivity, S_SortAlphabaticalList))
(NotDone, ActionControl, (I_MainActivity, S_SortUnfinishedList))
(DisplaySort, Resnponder, (S_DisplaySortedList))
(RemoveAll, ActionControl, (I_MainActivity, S_DeleteLists))
(AddList, ActionControl, (I_AddNewList))

MainActivity is
(SortingList, ActionControl, (I_SortMenu))
(AddList, ActionControl, (I_AddNewList))
(DsiplayLists, Responder, (S_DisplayLists))

EditList is
(MarkAll, ActionControl, (I_EditList, S_MarkAllBoxes))
(UncheckAll, ActionControl, (I_EditList, S_UncheckAllTasks))
(DisplayTasks, Responder, (S_DisplayTasks))
(ClearList, ActionControl, (I_EditList, S_DeleteAllTasks))
(RenameList, ActionControl, (I_Rename))
(AddTask, ActionControl, (I_AddTask))
(Back, ActionControl, (I_MainActivity))

AddNewList is
(OKButton, ActionControl, (I_MainActivity, S_AddList))
(CancelButton, ActionControl, (I_MainActivity))

AddTask is
(OKButton, ActionContrlol, (I_EditList, S_AddNewTask))
(CancelButton, ActionControl, (I_EditList))

RenameList is
(OKButton, ActionControl, (I_EditList, S_Rename))
(CancelButton, ActionControl, (I_EditList))
```

*Figure 22 PModel for To-Do List app.*

## 6.2 Presentation Interaction Model (PIM)

The Presentation Interaction Model (PIM) extends the PModel by mapping out the dynamic behavior of the user interface. It captures the sequence of state transitions triggered by interaction

behaviors, providing a comprehensive view of the application's flow. Each state in the PModel corresponds to a unique state in the PIM. Transitions between these states are labeled with the triggering I-behaviours, linking user actions to their resultant state changes. Figure 24 shows the PIM for the To-Do List app. The initial state in the app is the 'Main Activity' screen, which shows the different transitions that can be made from there. For example, adding a new list from the 'Main Activity' is linked to two other states with I-behaviour, while the last state transitioning from 'Add New List' to the 'Main Activity' is labeled with S-behaviour to present the newly created list.

The PIM serves as an outline for generating automated test cases, ensuring that all possible interaction paths are systematically explored. We can identify potential edge cases and robustness issues by modeling the UI's dynamic behavior. Figure 23 shows the PIM for the To-Do List application. It illustrates the stages of creating a new list, and the subsequent states represent the different stages of the interactions.

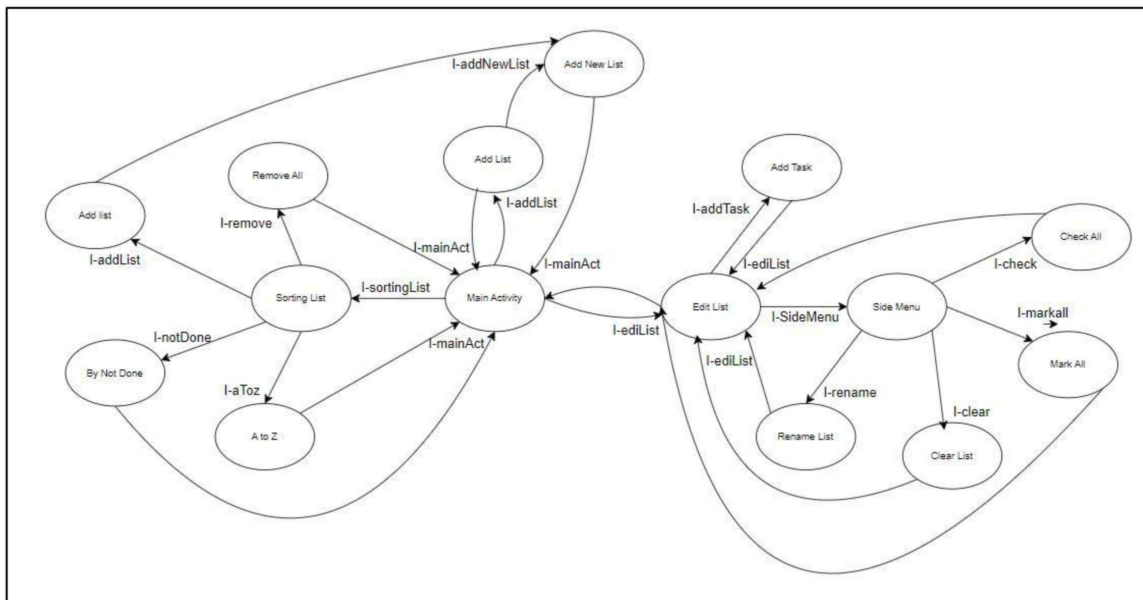


*Figure 23 PIM mapping for To-Do list app.*

## 6.3 Presentation Model Relation (PMR)

The Presentation Model Relationship (PMR) establishes a formal relationship between the UI's interactive elements and underlying system functionality. This model maps system behaviors (Behavior) in the PM to a specific operation in the application's formal specification. The PMR is essential in bridging the gap between UI actions and system responses, providing a rigorous

framework for verifying the application's correctness. It facilitates precise and reliable test generation by correlating user interactions with their expected outcomes.

## 6.4 Using PModels

This section explores the advantages of using Presentation Models (PModel) to represent an app's state and behavior in testing compared to traditional capture/replay tools like Espresso Test Recorder (ETR).

**Stability**

The generated scripts from ETR often break with any changes in the UI due to their dependency on the widget ID, text, and element hierarchy. These test scripts are less stable because their reliance on specific attributes means that any UI changes require manual modifications, leading to frequent breakages. For example, changes to a button will require updates to either its ID, text, or both. ETR's dependence on checking positions also contributes to the fragility of these scripts. For instance, the test in Figure 25 checks the positions of both the 'buttonPanel' and 'button1' and the button's attributes (text and ID).

In contrast, PModel captures the user's states and the widget IDs within those states instead of their positions. It also captures the transitions leading to clicking the 'OK' button rather than the hierarchical order. This approach means any UI changes can be applied to the model to generate new test cases, enhancing stability.

```
ViewInteraction appCompatButton = onView(
        allOf(withId(android.R.id.button1), withText("OK"),
                childAtPosition(
                        childAtPosition(
                                withId(androidx.navigation.ui.R.id.buttonPanel),
                                position: 0),
                        position: 3)));
appCompatButton.perform(scrollTo(), click());
```

*Figure 24 Validating the positions of elements for references.*

**Maintainability**

Considering the test scripts are easier to edit using PModel, they are easier to maintain since they cover the app's behavior, widget, and state transitions. As a result, the need for test script modification is minimized, making the tests more stable and easier to maintain. As the app improves, the PModel will be modified to mirror the changes applied to the app and new test scripts will be generated manually, reducing manual effort. In ETR, changes to the 'OK' button must be manually applied across multiple test scripts where the button was clicked or for the user to record a new test. In the figure above, the test script checks the button's position with the position of the parent state. Compared to the PModel, it would check the state of the button.

**Reusability**

PModel supports the reusability of test components. Standard states and transitions are defined once within the PModel, allowing reuse across different test scenarios. Once the user has mapped the states and the interactions linked to the application's elements in the model, they can be used across various test cases. This simplifies the generation of test cases and improves the efficiency of the testing process.

**Scalability**

As applications become more complex, managing test scripts manually becomes increasingly challenging. Using PModels allows for more straightforward modification and new application implementation without fearing breaking the existing test scripts. This flexibility provides comprehensive test coverage in complex and developing applications.

## 6.5 Summary

In summary, the interaction modeling framework presented in this chapter provides a comprehensive approach to capturing and testing UI behaviors in Android applications. By leveraging the principles of the PModel, PIM, and PMR, we can systematically generate and execute automated test cases using Espresso. This method enhances test coverage and ensures the robustness and correctness of the application under test.

# 7. PMed Tool

This section explores the PMed tool design using Presentation Models (PModels) to cover the ETR limitations and ensure sufficient testing. First, we will explore the design of test cases, illustrating the structured approach taken for ultimate test coverage and accuracy. Then, we will provide an interaction with the PMed tool walkthrough, detailing its interface, functionality, and step-by-step process of how the user generates test cases. This will give an understanding of the tool's capabilities in automated testing.

## 7.1 Test Case Design

The process of generating test cases for the PMed tool involves a structured approach that ensures thorough coverage and accurate support of the app's functionality. This begins with defining the specific scenarios that need to be tested. Scenarios are derived from the requirements and user interactions with the app, ensuring that all possible user behaviors are considered. The first step in test case design is the creation of a PModel. The PModel is an abstract representation of the app's UI components and interactions. The PModel is created using the PIMed tool, and each UI element is mapped to its corresponding actions, states, and transitions. This model acts as a design pattern for generating test cases, providing a clear and organized framework for understanding the app's behavior.

Once the PModel is established, the next phase involves specifying the criteria for test case generation. This includes defining the coverage criteria, such as ensuring all UI elements are linked with all possible states and transitions. Additionally, the accuracy and robustness of the test would be emphasized, ensuring that the tests not only cover all possible interactions but also do so reliably and can withstand changes in the application's UI without breaking. The tool's ability to convert the PModel into executable test scripts facilitates the actual generation of test cases. This conversion process involves mapping the abstract elements and interactions defined in the PModel with the exact elements in the app. These elements are extracted from the app's layout, where they are defined. The tool's design is to automate the generation of test scripts that can be executed in the Espresso framework to validate the app's functionality. This comprehensive approach to test case design ensures that the generated tests are thorough and maintainable, providing a solid foundation for automated testing of Android apps.

## 7.2 Tool Design

The PMed tool's design focuses on providing an intuitive interface for generating automated tests. This section provides a detailed walkthrough of using the tool, from setting up the environment to running the test cases.

1. Creating PModel

The first step before using the PMed tool is to create a PModel. This involves defining the UI components of your application and their interactions. The tool for creating models is called PIMed. It has a user-friendly interface where you can visually map out the elements of the app, such as buttons, text fields, and menus, along with possible user interactions like clicks, swipes, and text input. Each element and interaction is added to the PModel, manually named, and linked to other components, making building a comprehensive model of the app's UI easy. Figure 25 shows the interface of the PIMed editing the Presentation Model for the To-Do List app.

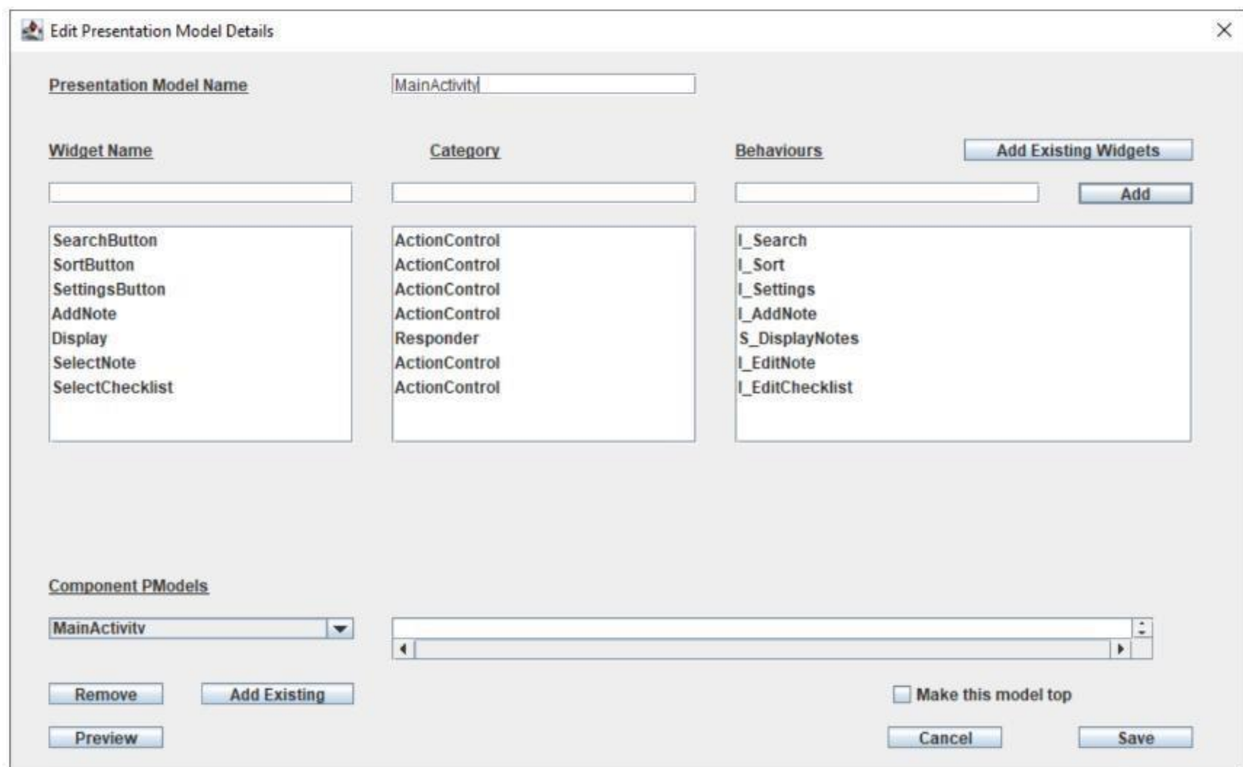


*Figure 25 Creating PModel using the PIMed tool.*

2. Defining Test Scenarios

Once the PModel is created, we need to import it into the PMed tool by clicking on ‘Load XML File’, Figure 26. Once the model is extracted, the elements can be mapped with the ‘Map Elements’ button, transferring the user to another window, as shown in Figure 27, which maps all the elements from the model with the app’s UI. After that, the user can define the test scenarios, which specify the sequence of interactions that need to be tested. By interacting with the tool and model’s elements, the user can create multiple scenarios and view the end-output of that specific scenario.

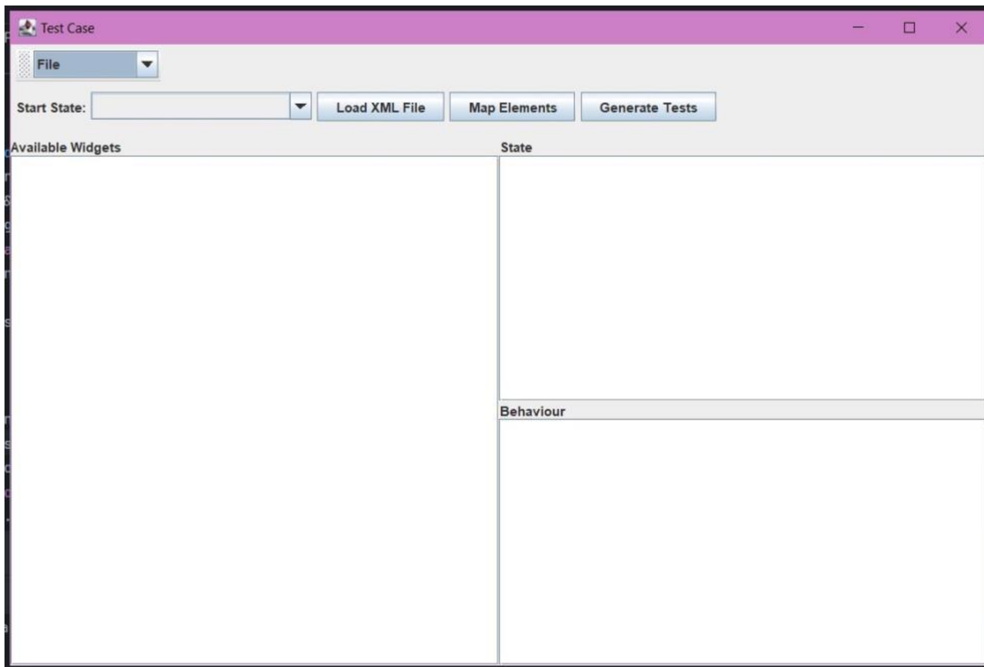


*Figure 26 PMed Interface*

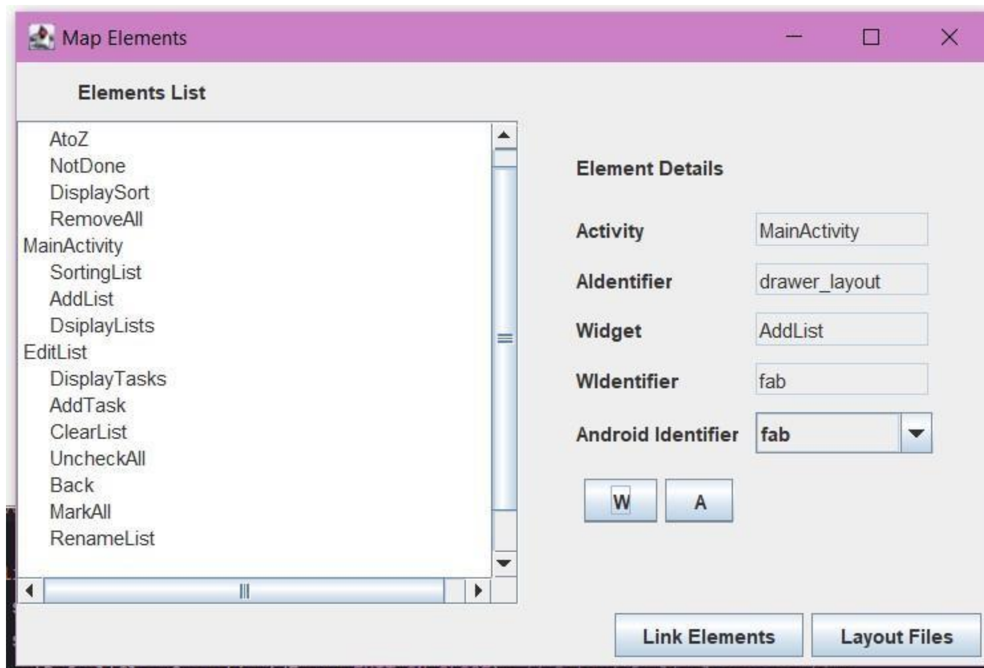


*Figure 27 The UI of Mapping Elements Feature*

Creating scenarios can be done using the following steps:

The user must select the scenario's preferred ‘Start State’ to present all the linked widgets and behaviors for the chosen state. The ‘Start State’ refers to the state or page in the application being tested, extracting its component in the PIM model. The start state is a drop-down menu showing all the states extracted from the model the user selects. This feature allows the user to view the transition for a selected widget from that state and its behaviors. It also allows the user to create a scenario starting from any state in the application. In

Figure 28, the selected state is ‘SortingList’.

Once the start state is selected from the dropdown menu, all the relevant widgets linked to that state will automatically appear in the widget list on the left side. The app's widgets include buttons, text boxes, Menus, etc. Different states will show different widgets—figure 28 lists all the widgets from the ‘SortingList’ state.

After the user selection of a widget, all the triggered states will appear on the ‘State List.’ The transition states the sequence that the selected widget would trigger, allowing the user to check and view the possible sequence. Figure 28 shows the transition of creating a new list from the

'Sorting List' state on the To-Do List app. Considering the app is not very complicated, the transition is short, where the user starts from the 'Sorting List' and ends in 'Main Activity' with only the 'AddNewList' state in between. In this feature, we look for the I_Behaviour widget component in the model. This allows the user to understand and view the start and end sequence of the scenario, as well as all the linked states to the sequence.

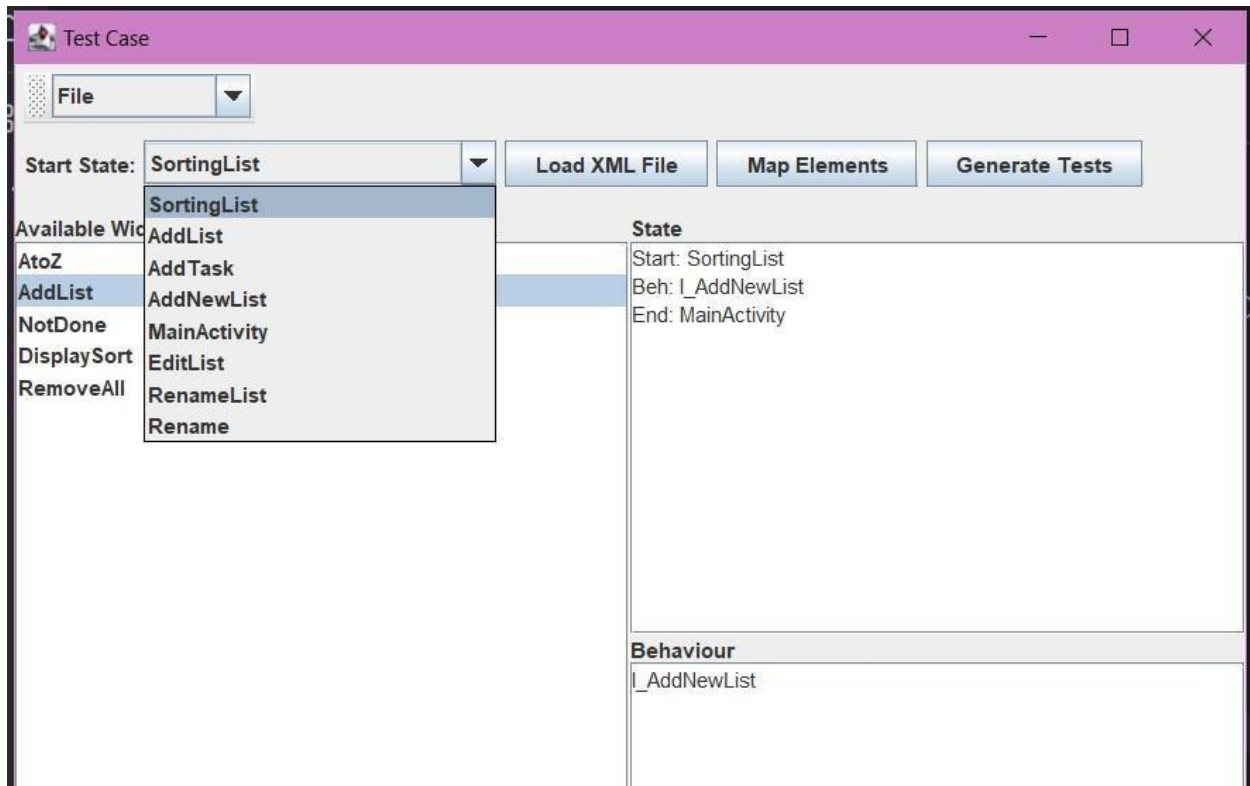


*Figure 28 Example of To-Do List Interaction with the PMed tool.*

3. Generating and Running Test Cases

After selecting the scenario to be tested, the tool is designed to generate the corresponding test cases automatically. The PMed tool converts the PModel and the defined scenarios into executable test scripts for Espresso. This involved translating the abstract elements and interactions of the PModel into Espresso test code. The user can export these test cases as an XML file to implement in Android Studio. Any changes to the application structure or elements should be applied to the PModel first, then editing the new PModel and regenerating new test cases.

# 8. Discussion

This section critically discusses the research findings, the challenges faced, and the implications of changing the research approach from implementing a tool to designing one. This shift was necessitated by time constraints, which impacted the original plan and led to a reevaluation of the project's scope and objectives.

## 8.1 Design versus Implementation

Initially, the aim was to implement a complete tool for automated capture/replay test generation using Espresso. However, the focus shifted to designing the tool due to time limitations. This change in approach had significant implications for the research.

### 8.1.1 Design Focus:

The design phase allowed for a more thorough conceptualization of the tool. The emphasis was placed on creating a robust theoretical framework that outlines the functionalities and integrations necessary for the tool's success in automatically generating test cases using PModels. This included specifying the widgets' interaction transition and behaviors and their integration with existing testing frameworks like Espresso. Designing the tool provided an opportunity to create a flexible template that can be adapted or expanded upon in future implementations. The design process included factors for scalability, user interface requirements, and potential integration with other tools or technologies.

### 8.1.2 Challenges in Implementation:

The implementation phase would have required overcoming significant technical challenges, including handling the complexities of Android's diverse ecosystem, dealing with different device configurations, and ensuring compatibility with several operating system versions. Implementing a fully functional tool within the given timeframe would have required considerable resources, including time, technical expertise, and possibly financial support for testing and debugging.

## 8.2 Limitations

The shift from implementation to design brought to light several limitations:

**Lack of Practical Validation:** A fully implemented tool made conducting real-world testing and validation of the proposed design possible. This limits the ability to evaluate the practical effectiveness and efficiency of the tool.

**User Feedback:** Implementing the tool would have allowed for evaluating test cases and making repeated improvements based on actual results. While the design phase was thorough, it needed to provide opportunities for proper investigation into implementing PModels to generate test cases for Android applications.

## 8.3 Future Work

The tool's design lays a strong foundation for future work, prioritizing the implementation of the designed tool, followed by extensive testing in diverse real-world scenarios. This will help prove the theoretical framework and identify unexpected issues during practical application. Also, enhancing the user interface to make the tool more user-friendly and accessible to testers with different backgrounds could improve its implementation and effectiveness. This includes providing comprehensive documentation and support for users. Future versions of the tool can incorporate advanced features such as AI-driven test generation and self-healing test scripts, which can further enhance the tool's capabilities and reduce maintenance overheads.

# 9. Concluding Remarks

This thesis explored the design and potential implementation of an automated test generation tool for Android applications using the Espresso framework. It focused on integrating model-based techniques to enhance test robustness and reliability. The transition from implementation to design allowed for a thorough conceptualization and theoretical framework, laying a solid foundation for future development.

## 9.1 Summary of Key Findings

### 9.1.1 Model-Based Testing Advantages

Model-based testing helps automate the generation of test cases, making the testing process faster and more efficient. It also allows for creating abstract models of the app under test, improving the test coverage. By automating test case generation and execution, model-based testing can reduce the overall testing costs associated with manual methods, reducing the process of the testing phase. Furthermore, it proves to be particularly effective for testing complex interactive applications. PModels in Android app testing provide a detailed representation of the app’s behavior and functionality, helping users understand how the app works. It also ensures that test cases closely align with the app's anticipated behavior, enhancing testing accuracy. Additionally, users can create scenarios starting from any state in the app using PModels, allowing for thorough testing from various starting points. Testing from different start states, especially for complex apps, makes it easier to manage and enable the testing of multiple states, simplifying the process of editing and maintaining the app’s changes. Thus, it allows more straightforward modification and implementation as the app develops, making test scripts adaptable to evolving apps without significant manual effort.

### 9.1.2 Tool Design and Evaluation

The tool's design contained insights from automated testing tools like BARISTA and Guider. These tools highlighted essential features such as structural and visual characteristics for GUI test script repair and capturing and replaying user interactions to generate reliable test cases. The proposed design included these features, improving the potential effectiveness of the tool.

### 9.1.3 Challenges and Limitations

The research identified several challenges, including testing and generating test cases after mapping elements and focusing on Android applications, which may limit the tool's applicability to other platforms. These limitations underscore the need for further development and validation in diverse real-world scenarios.

## 9.2 Implications for the Field

The implications of this research extend to both theoretical and practical domains. It highlights the importance of robust modeling techniques in reducing test fragility and improving automated test reliability. Practically, the proposed tool design offers a proposal for developing flexible and adaptable automated testing solutions for Android applications. The research features potential benefits for developers and testers, including reduced time and effort spent generating and maintaining test cases, eventually leading to more reliable app testing.

## 9.3 Future Work

Future research should focus on implementing the designed tool and conducting extensive validation in real-world scenarios. This will provide practical insights into the tool's effectiveness and identify areas for improvement. Enhancing the automation of the element mapping process and improving the user interface can make the tool more accessible and efficient for testers. Providing comprehensive documentation and support will further facilitate its adoption. Integrating the tool with continuous integration and continuous deployment (CI/CD) pipelines can streamline the testing process and ensure that automated tests are integral to the software development lifecycle. Advanced features such as AI-driven test generation and self-healing test scripts can further enhance the tool's capabilities. Research into machine learning techniques for predicting and adapting to UI changes can reduce maintenance overheads and improve test robustness.

# 10. References


[1] S. Chevalier, "Leading shopping apps for Android in the U.S. 2021, by daily active users." Statista.com. https://www.statista.com/statistics/1200832/google-play-shopping-apps-in-the-usbynumber-of-dau/ (accessed Jun 17, 2024).

[2] L. Ceci, "Number of available applications in the Google Play Store from December 2009 to December 2023." Statista.com. https://www.statista.com/statistics/266210/number-ofavailableapplications-in-the-google-play-store/ (accessed Jun 17, 2024).

[3] L. Ceci, "Leading Android apps worldwide 2023, by revenue." Statista.com. https://www.statista.com/statistics/271674/top-apps-in-google-play-by-revenue/ (accessed Jun 20, 2024).

[4] X. Yaun, and A. M. Memon, "Generating event sequence-based test cases using GUI runtime state feedback". *IEEE Transactions on Software Engineering*. Vol. 36, no. 1, pp. 81-95, (2009).

[5] J. Ferrer, F. Chicano, & E. Alba, "Estimating software testing complexity". *Information and Software Technology*. vol. 55, no. 12, pp. 2125-2135, (2013), doi:10.1016/j.infsof.2013.07.007.

[6] A. K. Jha, D. Y. Kim, and W. J. Lee, "A framework for testing Android apps by reusing test cases". *IEEE Xplore*, May 2019, https://ieeexplore.ieee.org/abstract/document/8817002.

[7] X. Yuan, M. B. Cohen, and A. M. Memon, "Towards dynamic adaptive automated test generation for graphical user interfaces". *CiteSeer X (The Pennsylvania State University)*, Jan 2009, doi: https://doi.org/10.1109/icstw.2009.26.

[8] Y. Lei, R. Kacker, D. Kuhn, V. Okun, & J. Lawrence, "IPOG/IPOG: Efficient Test Generation for Multi-Way Combinatorial Testing." *Software Testing, Verification & Reliability*, vol. 18, no. 3, pp. 125-148, Sep. 2008, doi: https://doi.org/10.1002/stvr.381.

[9] S. Keele, "Guidelines for performing systematic literature reviews in software engineering." *Technical Report*, Vol. 5, July 2007.

[10] M. Fatehah, V. Mezhuyev, and M. Al-Emran, "A systematic review of metamodelling in software engineering." *Recent Advances in Intelligent Systems and Smart Applications*. Pp. 3-27, Jun 2020.

[11] S. Singhal, N. Jatana, B. Suri, S. Misra, and L. Fernandez-Sanz, "Systematic literature review on test case selection and prioritization: A tertiary study." Applied Sciences, vol. 11 no. 24, Dec. 2021.

[12] A. Q. Gill, V. Behbood, R. Ramadan-Jradi, and G. Beydoun, "IoT architectural concerns: a systematic review." *Proceedings of the Second International Conference on Internet of Things*, *Data and Cloud Computing*. Mar. 2017.

[13] F. Ihirwe, D. Di Ruscio, S. Mazzinin, P. Pierini, and A. Pierantonio, "Low-code engineering for Internet of things: a state of research." *Proceedings of the 23rd ACM/IEEE international conference on model driven engineering languages and systems: companion proceedings*. Oct. 2020.

[14] J. Bowen, and S. Reeves, "Generating obligations, assertions, and tests from UI models." *Proceedings of the ACM on Human-Computer Interaction.* Pp. 1-18, June 2017.

[15] J. Bowen, and S. Reeves, "Modelling safety properties of interactive medical systems." *Proceedings of the 5th ACM SIGCHI symposium on Engineering Interactive Computing Systems*. June 2013.

[16] J. Turner, J. Bowen, and S. Reeves, "Model-based testing of interactive systems using interaction sequences." *Proceedings of the ACM on Human-Computer Interaction*. Pp. 1-37, June 2020.

[17] J. Turner, J. Bowen, and S. Reeves, "Simulating Interaction Sequences." *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. June 2018.

[18] J. Turner, J. Bowen, and S. Reeves, "SeqCheck: a model checking tool for interactive systems." *Companion Proceedings of the 12th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. June 2020.

[19] T. Xu, M. Pan, Y. Pei, G. Li, X. Zhang, T. Zhang, Y. Deng, and X. Li, "Guider: Gui structure and vision co-guided test script repair for android apps". *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. July 2021.

[20] J. Smith and A. Johnson, "Comparing the effectiveness of capture and replay against automatic input generation for Android graphical user interface testing". *Journal of Software Engineering*, vol. 10, no. 3, pp. 123-135, Oct. 2020.

[21] A. Machiry, R. Tahiliani, and M. Naik, “Dynodroid: An input generation system for android apps”. *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. Aug. 2013.

[22] S. Negara, N. Esfahani, and S. Buse, “Practical android test recording with espresso test recorder.” *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. Pp. 193-202, May 2019.

[23] J. Turner, "Supporting Interactive System Testing with Interaction Sequences". *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. June 2018.

[24] J. Turner, J. Bowen, and S. Reeves, “Using abstraction with interaction sequences for interactive system modeling.” *Software Technologies: Applications and Foundations.* pp. 25-29, June 2018.

[25] J. Bowen, and S. Reeves. “UI-design driven model-based testing”. *Innovations in Systems and Software Engineering*. pp. 201-215, May 2013.

[26] I. Arcuschin, C. Ciccaroni, J. p. Galeotti, and J. M. Rojas, “On the feasibility and challenges of synthesizing executable Espresso tests.” *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*. Pp. 92-102, May 2022.

[27] J. Bowen, and S. Reeves, “Modelling user manuals of modal medical devices and learning from the experience.” *Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems*. Pp. 121-160, June 2012.

[28] W. Lam, Z. Wu, D. Lie, W. Wang, H. Zheng, H. Lou et al., "Record and replay for android: Are we there yet in industrial cases?". *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. Pp. 854-859, Aug. 2017.

[29] M. Fazzini, E. N. D. A. Freitas, S. R. Choudhary, & A. Orso, “Barista: A technique for recording, encoding, and running platform independent android tests.” *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. Pp. 149-160, Mar. 2017.

[30] R. Coppola, M. Maurizio, and T. Marco, "Mobile GUI testing fragility: a study on open-source android applications." *IEEE Transactions on Reliability.* Vol. 68, no. 1, pp. 67-90, March 2018.

[31] J. Bowen, and S. Reeves, “Combining models for interactive system modelling”. *The handbook of formal methods in human-computer interaction*. Pp. 161-182, 2017.