MASTER

mTESTAR for scriptless GUI testing on Android and iOS applications

Jansen, Thorn

Award date:
2021

mTESTAR for scriptless GUI testing on Android and iOS applications

Eindhoven, September 29, 2021

This report counts 67 pages

Author:
Thorn Jansen | 1003562

Supervisors:
Luo Yaping (TU/e)
Kevin van der Vlist (ING)
Robbert van Dalen (ING)
Jeroen Keiren (Comittee Member)

Other Supervisors:
Pekka Aho (Open Universiteit)
Tanja Vos (Open Universiteit)

Mathematics and Computer Science Department

M.Sc. Computer Science and Engineering
Eindhoven University of Technology

Abstract

Software development is a very large industry where trillions of dollars are being spent. At the same time, a lot of this money is lost due to high failure costs. Software testing aims to reduce the waste of resources. GUI testing is a crucial aspect of the overall testing process due to its ability to test end-to-end and the GUI being the user facing component of the software. Traditionally, manual tests and scripted automated tests are used to determine if the GUI meets the requirements. However, both approaches are very costly in terms of both money and time. Scriptless testing attempts to address the costs associated with GUI testing. With a rapidly growing mobile application market, there is an opportunity to apply scriptless testing. However, scriptless testing is not well-established in this area. Therefore, this study presents a tool for scriptless GUI testing in the mobile domain.

We present a literature study on the available tools and the failure detection component (oracle) for automated testing. We extract several design aspects important for creating a scriptless GUI testing tool. Additionally, we obtain an overview of testing oracles. Next, we present the mTESTAR tool for scriptless GUI testing in the mobile domain and the implemented oracles. Lastly, we validate mTESTAR on the industrial ING Bankieren application. From the validation, we determine that mTESTAR outperforms two state-of-the-art tools and achieves similar performance as the scripted testing on ING Bankieren. Overall, mTESTAR is to be used in combination with scriptless testing to provide maximum value for the testers.

Acknowledgements

First of all I would like to thank Yaping Luo, Robbert van Dalen, and Kevin van der Vlist for their time and valuable input for my graduation project. Their input throughout the weeks and presence in the weekly meetings has been crucial to completing this project.

Next, I would like to thank Pekka Aho and Fernando Pastor Ricós for their help with TESTAR and interesting suggestions throughout the project.

Finally, I would like to thank my family for their continuous support throughout my studies.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

IT is a large global industry. Global IT spending will grow to $4.4 trillion in 2022, of which 15% is spent on software development [1]. At the same time software development has high failure costs. A lot of resources go to waste due to failing software, studies estimate the damage worldwide to be approximately $3 trillion [2]. Software testing is part of the software development process aimed at lowering the waste of resources. To do so, testing validates the software quality, software security, and software reliability. Ideally, by validating these three attributes software testing ensures software is error or bug-free. However, software testing cannot ensure the absence of errors and bugs, it can only determine the presence of errors. Therefore, validating quality, security, and reliability only allows software testing to determine if requirements are met. Meeting the requirements lowers the waste of resources.

According to Mike Cohn, the tests on software consists of three types; unit tests, service tests, and Graphical User Interface (GUI) tests [3]. Unit tests focus on testing small individual components, service tests verify the quality of a combination of the small individual components, and GUI tests focus on end-to-end testing of the whole application under test (AUT). GUI tests are a critical aspect of the whole software testing process. It is the only type of test which tests the AUT end to end, validating the quality of all components of the application together. Additionally, the GUI is the user facing part of the software application. Having it meet the security, quality and reliability requirements is essential as it directly affects the user of the AUT.

1.1 A closer look at automated GUI testing

Traditionally, there is manual GUI testing and automated GUI testing [4]. In manual testing humans interact with the available GUI to determine if the GUI meets the requirements. In automated testing test sequences are created by developers to explore and repeatedly test the GUI, this is also called script-based testing. In each of these test sequences it is checked if the requirements for quality, security, and reliability are met. Manual testing has the ability to leverage the domain specific knowledge the human tester possesses, while automated scripted testing supports better repetition support. However, both manual and script-based testing have the important weakness related to cost. It is very expensive to repeatedly let humans explore the application under test [5]. Although script-based testing lowers the resources required compared to manual testing, there is still a significant cost to maintaining the test scripts [4]. Additionally, script based testing no longer has direct involvement of human testers and therefore the challenge of integrating domain specific knowledge. Lastly, script based testing struggles to mimic the random behavior of users. Overall, this results in an expensive (up to 50% of the development costs [6, 7]) and challenging road to verifying if the GUI meets the requirements. This has also been confirmed by a survey held under a number of software developers active in the industry.

Scriptless testing attempts to address the challenges present in manual and automated GUI testing. This is a process in which the tests are both generated and executed completely automatically, saving resources and time. Algorithms are used to achieve the automated generation of test sequences. Due to the algorithms generating the test sequences, the resources consumed by scriptless testing are lower than for manual or scripted testing. Additionally, scriptless testing allows for randomness in the generating of test sequences. Therefore it addresses the limitation of scripted testing concerning mimicking the random behavior of users. A tool that provides a scriptless GUI testing platform is TESTAR [8]. TESTAR is open source, providing scriptless testing for web and desktop applications. TESTAR's approach and implementation have been industrially verified while also saving time and resources compared to automated scripted testing [9–11]. Thus, proving that scriptless testing can benefit the development of software.

1.2 Automated GUI testing for the mobile domain

In recent years the trend of "Mobile First" is gaining significant traction [12]. This is the trend where software is first designed for mobile and progressively enhanced for larger screen real estate (e.g. desktop and web application). This leads to a rapidly-growing mobile application market [13, 14], specifically for Android and iOS applications. The growing mobile application market, together with the Mobile First trend, creates an important opportunity to design a scriptless testing method for mobile GUI testing in an attempt to lower resource needs.

Looking at scriptless testing for mobile applications there are a few challenges that need to be addressed before it can effectively be applied in an industrial setting. Most notably, although gaining traction, scriptless testing is not yet well established in the mobile application development industry [15]. Tools like TESTAR for desktop or web application scriptless testing are not applicable to mobile. The majority of the android scriptless GUI testing tools available are academic tools, that are neither well maintained and established, nor properly verified on industrial-grade software applications. While for iOS applications, to the best of our knowledge, there do not exist any scriptless testing tools.

A notable second challenge is the oracle problem [16,17]. Oracles are components that distinguish whether the AUT has quality issues or meets the requirements specified [16]. There is little work available on oracles for automated testing and even less for scriptless testing [17]. However, test oracles significantly contribute to test effectiveness and reduction of costs [18]. Therefore, it is valuable to create oracles that are effective for scriptless testing.

Both challenges together provide an opportunity for developing a scriptless testing tool for Android and iOS applications while simultaneously integrating an oracle improvement for determining whether the AUT meets the requirements. Additionally, it is important to validate the scriptless testing tool on an industrial case to ensure the tool is applicable in industry.

1.3 Research questions

In this research the main goal is to develop a tool for scriptless GUI testing in the mobile domain. We make use of the knowledge and insights that are currently available to give direction to the development of the tool. Leading to the following research question guiding the work in this paper:

RQ: How can we apply scriptless GUI testing for mobile applications in an industrial environment?

TESTAR is open source tool and has promising approaches for test sequence generation available. It has been decided to extend TESTAR for mobile applications. It will be validated on an industrial use case of the ING bank. This leads following sub research questions being extracted:

RQ1: What is the state of the art in the scriptless GUI testing for mobile applications?

RQ2: How can we extend TESTAR for either or both iOS and Android mobile applications in an industrial environment?

RQ3: What types of testing oracles can be used for mobile applications and how?

1.4 Research design

The research design employed in this thesis is design and action research [19]. To understand the state of the art in mobile GUI testing and understand oracles for scriptless GUI testing we perform a literature analysis. We analyze the design of the most important testing tools and determine the important design aspects. Based on this, we develop a tool called mTESTAR for scriptless mobile GUI testing. Additionally, we describe how we approach the oracle problem in mTESTAR. To validate the design and implementation presented, the performance of mTESTAR is measured on an industrial application, the Android ING Bankieren application, and compared to scripted testing and two state of the art mobile scriptless testing tools.

The remainder of the paper is organized as follows. Chapter 2 describes the background information and the related work. Chapter 3 presents the engineering work performed to develop a scriptless testing tool for the mobile domain. Chapter 4 introduces experiment setup, results and a discussion of the results. Chapter 5 discusses the threats to the validity of the experiment and proposes possible directions for future work. Finally, chapter 6 concludes.

This graduation project is realized at the Software Engineering and Automation chapter of the OmniChannel API Platform at ING[1] with support of the Open Universiteit[2]. Moreover, the graduation project is part of the Industrial-grade Verification and Validation (V&V) of Evolving Systems (IVVES) project. IVVES is an international project that connects 26 partners from 5 different countries aiming to improve AI-based approaches to achieve robust and comprehensive, industrial-grade V&V. Through, for example, machine-learning for control of complex, mission-critical evolving systems and services covering the major industrial domains in Europe [20].

[1]`https://www.ing.com/`
[2]`https://www.ou.nl`

Chapter 2

Background and related work

In this chapter the background material and related work for the topic of scriptless GUI testing for mobile applications will be described. The related work is divided into five subsections; the need for automated GUI testing, GUI exploration algorithms, oracles for automated GUI testing, introduction to the GUI scriptless testing tool TESTAR , and tools for scriptless GUI testing in for mobile applications.

2.1 The need for automated GUI testing

The GUI is an interface that allows users to interact with applications through graphical elements. As the GUI is the main point of entry for most users it is important to prevent failures and have the software respond as expected. To prevent these failures testing is applied, including testing the GUI. However, Alégroth *et al.* state that testing can take up to 50% of the development costs. In an attempt to lower the testing costs Alégroth *et al.* suggest automated scripted GUI testing can help [4]. Alégroth *et al.* validate this suggestion by using automated testing at Saab and Siemens. In both companies automated scripted test cases have been developed and a greater return on investment (ROI) can be observed compared to the manual testing approach.

Patel *et al.* evaluated if random scriptless testing, also called monkey GUI testing, is more effective than manual testing in the setting of Android applications [21]. They concluded, after empirical analysis of 79 applications, that manual testing and monkey testing achieve very similar code coverage. However, Patel *et al.* note that the costs for monkey testing are lower than the costs for manual testing.

Vos *et al.* discuss the scriptless testing tool TESTAR [22]. In their work, they claim that there is an increasingly widespread use of incremental processes and continuous integration for software testing. To use these processes effectively and successfully scriptless testing is a requirement. The reason mentioned for requiring scriptless testing is the decrease in available quality assurance time for each release, combined with the increasing level of complexity of software systems. Although it is not specifically mentioned to what degree scriptless testing can help the overall process, Vos *et al.* determined it essential to maintain the current day CI processes. Furthermore, scriptless testing is posed to lower the high maintenance costs associated with automated testing. Potentially improving the resources saved even more.

2.2 Scriptless GUI exploration algorithms

A scriptless testing tool automatically generates the test sequences for the AUT. To do so, it needs an exploration algorithm (also called the action selection algorithm) to generate the events for each test sequence. Several algorithms are used for GUI exploration for scriptless testing tools. To understand scriptless testing better, we introduce the most common algorithms.

- **Random exploration** [23] is a basic exploration algorithm. It is often also called monkey testing [24] as it mimics the behavior of a monkey using a computer. Without optimizations or improvements, the concept is simple; it arbitrarily takes one action out of all possible actions for the current state of the GUI.

- **The unvisited action first algorithm** is a simple algorithm improving upon the random exploration algorithm. Unvisited action first selects actions at random, but actions that already have been executed are not considered. The idea is that actions already executed do not lead to new GUI states.

- **Breath-First-Search** is an algorithm aimed at systematically exploring the GUI of the AUT [25]. In BFS all actions of the current GUI state are executed, and the resulting GUI states are saved on a stack. Once all actions have been executed the top state is pulled from the stack and the process starts over. The exploration continues until the whole GUI has been explored.

- **Depth-First-Search** is also an algorithm aimed at systematically exploring the GUI of the AUT [26]. In DFS once an action has been found it is executed. This continues until there is a GUI state without actions, or an already discovered GUI state is encountered. The algorithm then proceeds to revert one state and execute the second action. DFS continues until no more new GUI states are discovered.

- **Supervised learning** is a technique based on learning [27, 28]. In supervised learning, a dataset with actions executed by users on similar applications as the AUT is required. Using this dataset the algorithm learns how to interact with applications. Once done learning, the algorithm received the AUT, and it decides for every GUI state which action to take.

- **Q-learning** is a reinforced learning approach [29]. In Q-learning, the algorithm is first allowed to investigate the AUT. From this investigation stage it learns which (type of) actions lead to new states. Next is the the exploration stage. During exploration actions leading to new states are worth a reward. Q-learning tries to maximize the reward obtained and therefore tries to explore the AUT aiming to find new GUI states.

- **Genetic algorithms** [30] mimic the natural process of evolution; improving over time in an attempt to reach an optimal solution. For GUI exploration this means that several test sequences are generated for which the performance is measured. The genetic algorithm proceeds to slightly change each sequence. After the evolution of the test sequences, the performance is measured again. From the difference in performance, the genetic algorithm learns and proceeds to evolve all test sequences. Once the genetic algorithm finishes, a set of different test sequences exploring the GUI is obtained.

- **System action generation** [28] explores the AUT randomly. For each GUI state encounters it generates a number of system actions (orientation rotations, calls, moving application for foreground to background, ect.) and adds them to the test sequence. If already visited GUI states are encountered no system actions are added to the test sequences.

2.3 Oracles for automated GUI testing

Oracles are components that can distinguish between correct behavior and incorrect behavior for the AUT [16]. Due to this, test oracles used during testing significantly contribute to test effectiveness and reduction of costs [18]. Therefore, it is important to evaluate the existing literature on automated and scriptless GUI testing oracles.

When looking at test oracles there are two ways we can classify the oracles:

- Online/ offline oracles
- The type of oracle

Online oracles are oracles that run when testing the application. During the execution of the test sequence, the oracle continuously verifies whether the AUT is in a no-fault state. Offline oracles [31] are oracles that are applied after the test sequences have completed running, e.g. validating the state model discovered by the test sequences. For the type of oracle four categories can be distinguished [16, 32].

- **Specified oracles:** An oracle that leverages the formal specification of the AUT, where the formal specification defines what behavior is acceptable and what behavior qualifies as fault-state. For example, a predefined state model where all transitions are defined. If an action results in a different transition than specified in the state model, a fault has occurred.
- **Derived oracles:** An oracle that determines the correctness of an application based on the information derived from artifacts (e.g. documentation, system executions) or other versions of the AUT. For example, requiring multiple executions of an test sequence to conform to the first result obtained when executing the specific test sequence.
- **Implicit oracles:** An oracle that relies on general or implicit knowledge to determine if a application is in a fault-state or not. For example, if the AUT is no longer running a fault has occurred.
- **(Reduced) human oracles:** These are not automatic oracles but rather support systems to make it easier for humans to determine the correctness of the AUT. For example, providing an overview of the differences in output between multiple runs of the same test sequence.

Specified oracles are generally not preferred for scriptless testing. In specified oracles, it is required to have a formal specification of the application under test. When the application changes, this formal specification requires maintenance to correctly model the AUT. Essentially making use of specified oracles will move the maintenance burden from the scripts in testing to the oracle but does not address the overall maintenance cost problem. Note that there are exceptions for oracles based on invariants or very specific system properties. In these specific cases, specified oracles can be leveraged in scriptless GUI testing and require limited domain knowledge to be inserted into the oracle.

Derived oracles all work on the same principle; consistency or agreement between program executions. Examples of derived oracles are the following:

- N-version programming [33] which evaluates whether independent implementations of the same program return the same outputs.
- Metamorphic relations [34] which focuses on whether the same test sequence consistently returns the same output.
- Regression testing [35] focusing on whether different versions of the AUT behave the same.

- Inferring models from system execution and then using these models as a formal specification of the AUT [36].

What all these derived oracle techniques have in common is that they are looking for agreement or consistency between several program executions. The problem for derived oracles is their tailoring for one application. Overall, derived oracles require a lot of domain knowledge to be useful and therefore cannot be used by scriptless testing tool for generic use.

Implicit oracles are extremely suited for scriptless testing. Implicit oracles by definition are oracles that require no domain knowledge nor a formal specification to implement, and they apply to almost all programs [16]. This makes it possible to define generic oracles in scriptless testing tools which can be reused for different applications and requiring little maintenance if an application changes over time.

Human oracles are not the desired online oracles for scriptless testing. In the case of online oracles, they are infeasible as it would require a human to interact with the scriptless testing tool after each action taken on the GUI. In the case of offline oracles they can be used for helping algorithms learn or providing tables and graphs to the tester.

Note that with any type of oracle it is important to record false positives such that they are filtered out after one occurrence. As this list requires maintenance no oracle truly avoids maintenance costs.

Atif Memon is an active researcher in the area of GUI oracles [37]. He presents multiple different approaches to GUI oracles in his papers. Memon *et al.* defines GUI oracles to consist of two components; oracle information and oracle procedure [18]. Oracle information is the component that holds the information the oracle requires for comparing actual output and expected output. Oracle procedure is how the comparison takes place. Using these two building blocks Memon introduces different types of oracles in his (co-)authored papers. The first suggestion he makes is an improvement on the “golden version” oracle, a derived oracle. In the ”golden version” oracle a version of the application is taken and assumed to be perfect. Now when tests are performed the output is compared to this ”golden” version and when there is a difference the test has failed. The critical weak point is that obtaining a *golden version* of the application under test is not realistic. To address this issue Memon *et al.* suggest taking multiple versions and compare the results [38].

A second oracle suggested by Memon *et al.* is making use of a decision table, where the table represents a simplified model of the application under test. One axis is used for conditions and the other axis is used for actions. When certain conditions hold, it can then be determined from the table which actions are possible [38].

Formal modeling is a more extensive method of creating oracles. Memon *et al.* suggest building a model of the GUI under test. In test cases the resulting states are compared to the expected states according to the model, if there is a mismatch the test is marked as failed [38]. Memon *et al.* suggest using AI planning to support the model generation. In different papers Memon describes different approaches for creating the formal model of GUIs [39–41].

2.4 TESTAR a tool for scriptless GUI testing

A tool for the scriptless testing of desktop and web application is TESTAR [8]. TESTAR is an open-source tool that automatically generates and executes test sequences on an application under test (AUT). The set of possible actions on the AUT are based on the derived structure of the GUI at the time of testing. The goal of TESTAR is to detect violations of system requirements. These violations can be recognized through system crashes or error messages, or through user specified oracles. TESTAR achieves its functionality on the desktop and web applications by leveraging the accessibility application programming interface (accessibility API) for desktop applications [42], or the Selenium Webdriver [43] for web applications. For desktop applications, the operating system accessibility API allows TESTAR to retrieve the current structure and component information of the AUT. At the same time, the OS accessibility API is also used for executing actions on the AUT. For web applications, the same principle holds but instead the Selenium Webdriver is used as accessibility API. For both desktop and web applications, visualizations are drawn at the OS level of the host machine.

As not all applications work the same, it is valuable to support entering domain knowledge into TESTAR without requiring much maintenance of this domain knowledge. To support domain knowledge, TESTAR has customizable components, called protocols, in which the tester can add information or settings suited to the specific application under test. For example, the tester can customize which exploration algorithm is used, or which behaviors are classified as bugs.

2.4.1 General execution flow TESTAR

In Figure 2.1 the general execution flow of TESTAR is displayed. In more detail, the steps can be described as follows:

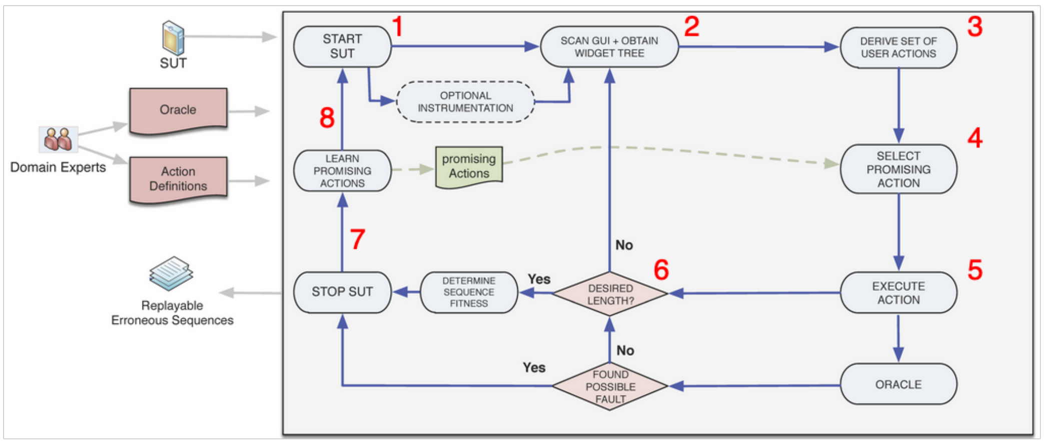


Figure 2.1: TESTAR general execution flow. Taken from `testar.org`

Step 1: Start the application under test (AUT) and wait until it is ready for interaction.

Step 2: Inspect the AUT to obtain information about the application and the individual components present in its current state. As a result a tree with the structure of the current state of the application is obtained, so-called widget tree.

Step 3: From the obtained widget tree determine which actions can be taken on the AUT.

Step 4: From the derived list of actions possible, select one specific action. Which action is selected depends on the action selection algorithm TESTAR is run with. By default TESTAR allows for random selection, takes an action previously unvisited, or uses the reinforced learning approach, Q-learning, to determine an action [44].

Step 5: Execute the selected action by interacting with the AUT.

Step 6: If no fatal faults are found steps 3,4, and 5 are repeated until the desired sequence length has been generated. The resulting sequence is evaluated and the AUT is stopped.

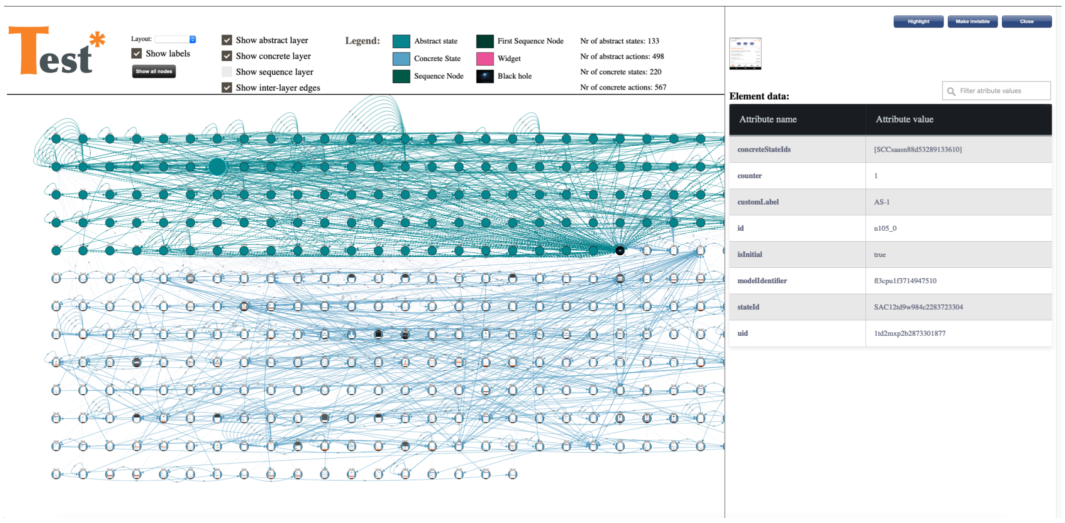
Step 7: If the desired number of sequences has not be reached yet the whole process is repeated from step 1.

Step 8: TESTAR finishes and exits gracefully.

2.4.2 State and state model

TESTAR works with the concept of state. At each point in time the GUI of the application under test is in a certain state. TESTAR uses this definition of state to create a state model of the GUI which helps with action selection. TESTAR identifies the state of the GUI under test by applying a hash function to all attributes of each widget on the current visible GUI, this result is called the concrete state. However, due to the vast number of widgets and the vast number of attributes of each widget usually present on a GUI state, the state model using the concrete state suffers from the state explosion problem [45]. Therefore, TESTAR abstracts and looks only at certain attributes of widgets to quantify a state, this is called the abstract state. In TESTAR the tester determines which attributes are used in the definition of the abstract state.

To store the states and state model, TESTAR makes use of the graph database OrientDB [46]. OrientDB is a multi-model open-source graph database that supports schema-less, schema-full, and mixed-schema modes. In the case of TESTAR, nodes in the Orient database correspond to the abstract states (which in turn represent states of the GUI) and transitions correspond to actions that resulted in the switch of the abstract state. For an example of how the state model stored in OrientDB is visualized see Figure 2.2.



Nr of abstract states: 133
Nr of abstract actions: 498
Nr of concrete states: 220
Nr of concrete actions: 567

Element data:

Attribute name	Attribute value
concreteStateIds	[SCCsaasn88d53289133610]
counter	1
customLabel	AS-1
id	n105_0
isInitial	true
modelIdentifier	fl3cpu1f3714947510
stateId	SAC12td9w984c2283723304
uid	1td2mxp2b2873301877

Figure 2.2: Example of how the state model of TESTAR can look visually.

2.4.3 TESTAR oracle

As TESTAR tries to test applications to determine if there are any faults present, it is vital that TESTAR can detect these faults. To achieve the detection of faults oracles are used. Currently, TESTAR supports an implicit oracle and a specified oracle; the implicit oracle checks if the application is still open and responding, also called the crash oracle. The second oracle is a specified oracle, it checks for for suspicious strings. The suspicious strings oracle allows for the specification of regular expressions that are checked in each string that has been found in the state. If it detects any of the suspicious regular expressions to be present it will report and exit. It is possible in TESTAR to program your own oracle in the source code, allowing testers to add domain knowledge to oracles within TESTAR. For the state of the art in oracle development see section 2.3.

2.5 Mobile scriptless GUI testing tools

Although scriptless GUI testing is not well established in the industry there exist a number of academic (Android) scriptless GUI testing tools. To understand the scriptless testing methods used by these tools it is important to investigate these tools. In this subsection a number of tools with the most interesting scriptless testing approaches will be introduced. Each method will be described on;

- Its overall approach to scriptless testing.
- The exploration algorithms used.
- The approach to understanding the GUI presented.
- The tools's flexibility to add application-specific information of the AUT.
- The tool's overall strengths and weaknesses

Before introducing different scriptless testing tools it is important to highlight the UI/Application Exerciser Monkey[1] [21]. This tool is built into the Android development environment and allows for random gestures on an application. This tool is often used as a benchmark of performance for scriptless testing tools to compare to. The reason for its function as benchmark is because of its availability. Beyond randomly sending touch events to the Android mobile device it does not possess any unique attributes.

Eskonen *et al.* introduced a tool that uses image-based deep reinforced learning to achieve scriptless testing [47]. Although originally designed for web applications, the authors claim the tool can easily be applied to other platforms as well due to the concept of image-based reinforced learning. What makes this tool unique is that it understands the GUI under test through screenshots, it does not interact with accessibility APIs or the OS of the device the AUT runs on to obtain information of the widgets on the screen. Instead, the concept of widgets is completely ignored by Eskonen *et al.* The reinforced learning algorithm tries to maximize the number of states visited, where non-identical screenshots are different states. Once the reinforced learning algorithm receives the input screenshot, it computes a probability on every pixel, the pixel with the largest probability is the location of the screen that should be interacted with. The tool by Eskonen *et al.* does not allow for modification to add application-specific AUT information. Additionally, it does not use test oracles. The core strength of the tool is that it needs very limited access to the AUT, as screenshots are the only requirement. Additionally, if the reinforced learning algorithm has a well-designed cost function and rewards, it has the potential to efficiently explore the states of the AUT. Overall, the authors claim that the reinforced learning tool performs significantly better than random scriptless testing tools.

Mao *et al.* designed a tool for systematic/ search -based scriptless testing on Android called Sapienz [48]. Sapienz's approach is to create test sequences that systematically explore the AUT. To improve Sapienz performance, genetic evolution algorithms are applied to the sequences created to get an even more

[1]`https://developer.android.com/studio/test/monkey`

diversified set of sequences. At the same time, Mao *et al.* try to minimize the sequence length while maximizing the code coverage and faults found. This stems from the belief of the authors that short sequences are more likely to occur in practice and thus are preferred over longer sequences. Sapienz original code is publicly available, however, Facebook has acquired Sapienz in 2017 and has since been for Facebook's internal use only, unobtainable for interested parties outside Facebook. The original Sapienz code adjusts its way of working based on the availability of the AUT. When the source code is available it is possible to examine the code at a closer level and get more coverage information at runtime. When the code of the AUT is not available, repackaging is attempted to achieve the code coverage at runtime. Lastly, Sapienz allows for integrating application-specific knowledge of the AUT. If desired, the developer can add certain actions which should be executed in sequence once certain, tester specified, conditions hold. Overall, the strength of Sapienz is its ability to create a diversified set of sequences that have the potential to cover the AUT extensively. The weakness of Sapienz is the testing oracle, as it only checks for application crashes.

Azim *et al.* have designed an Android systematic scriptless testing tool called A3E [49]. Azim *et al.* concluded from a user study that manual testing does not adequately cover Android applications. Their approach to solving this problem is a scriptless testing tool that builds a control flow graph from the byte code of the Android application under test. This means states and events changing the state of the application are modeled without the source code of the AUT. Through the bytecode A3E captures the data of widgets present in each GUI state. Once the event graph is obtained A3E offers two modes for generating test sequences; Targeted exploration tries to generate test sequences covering the different activities of the Android AUT and Depth-first-search exploration, which uses the same concept as depth-first-search to explore the AUT. At the time of writing, A3E outperformed the Android monkey tool concerning code coverage. Furthermore, A3E does not seem to support adding any application-specific knowledge to the exploration strategy and does not use an oracle. A3E's strong point is that there is more than one exploration strategy available. Unfortunately, A3E has not been maintained since 2013 so is unlikely to work on the latest Android releases and it is unclear if it still performs similar to current day scriptless testing tools.

Li *et al.* have designed a model based Android scriptless testing tool called DroidBot [50]. DroidBot builds a model of the AUT at runtime. The tester proceeds to select or implements its exploration method for this model to generate test sequences. DroidBot's interesting feature is that it does not require source code access and works on applications that can not be instrumented (encrypted applications) as it builds the model of the AUT at runtime and not through static (byte)code analysis. DroidBot has three main built-in exploration methods; depth-first, random, and unvisited action first. In each of the algorithms the performance is measured through code coverage achieved. DroidBot uses the Android UIAutomator[2] for determining the GUI hierarchy tree. As DroidBot allows developers to create their exploration strategy (or edit the existing strategies), DroidBot offers great support for injecting application-specific information of the AUT. Overall, DroidBots strengths stem from its flexibility to adjust for specific applications, while this is also its weakness as the tester cannot expect the best performance with the default exploration methods. Lastly, DroidBot does not look at the correctness of the GUI states, thus lacks an oracle component. *Li et al.* released DroidBot as an open-source tool and at the time of writing DroidBot is still actively maintained, making sure DroidBot works on the latest Android releases.

Li *et al.* have continued developing Android scriptless testing tools and have designed an extension on top of the DroidBot tool called Humanoid [27]. The tool is specifically designed to provide another exploration method for DroidBot. Humanoid is an image-based supervised deep learning method for exploring and creating Android test sequences. Humanoid works based on screenshots of the AUT. It is trained on a large dataset of human Android application interactions. From this dataset the deep learning model

[2]https://developer.android.com/training/testing/ui-automator

learns how humans usually interact with Android applications. When learning and passing screenshots of the AUT to Humanoid are done, the location the deep learning model would click, type, swipe, or long-click for each screenshot are returned. Humanoid replaces the exploration method of DroidBot and thus uses DroidBot its functions for interacting with the AUT. One of the core problems with Humanoid is that it is a big challenge to keep an up-to-date dataset from which the deep learning model can learn how users interact with Android applications. Additionally, the applications in the training set do not need to be representative of the AUT. Humanoid itself is also open source but has not been maintained since 2019.

Stoat is an Android stochastic model-based scriptless testing tool designed by Su *et al.* [51]. The approach used by Stoat contains both a dynamic and static code analysis component to create a model (finite state machine) of the GUI of the AUT. The dynamic analysis is a weighted GUI exploration in the sense that exploration is focused on actions that have a promise to increase code coverage. The model created contains the different states and transitions of the GUI that Stoat managed to extract. Once the model has been created Stoat uses a stochastic probability model to create sequences maximizing the code coverage of the application under test. In essence, the transition probabilities are semi-randomly mutated to get unique test sequences. Using this technique, Stoat outperformed A3E, UI/Application Exerciser Monkey, and Sapienz in terms of code coverage and detected crashes. Stoat needs code coverage information about the AUT to optimize its dynamic exploration, therefore Stoat has build in coverage measuring capabilities. For closed source tools ELLA [52] is used. For open-source tools EMMA [53] is used. Using both these tools allows Stoat to work on both closed and open-source applications. Stoat is released as an open-source tool but has not been maintained since early 2020. Additionally, while being open source it has no flexibility for injecting application-specific information about the AUT without drastically editing source code. Overall, Stoat strong point is its unique two stages approach of first creating a model followed by the approach of stochastic genetic mutation to generate the test sequences. The downside of Stoat is its inflexibility to adjustments for specific AUT's, its reliance on outside tools (EMMA and ELLA) for optimal performance, and having limited oracle support as it only looks at crashes of the as fault behavior.

Borges *et al.* created an Android monkey scriptless testing tool called DroidMate [54]. DroidMate sits between the Android OS and the AUT and uses the native Android UIAutomater tool to retrieve the GUI hierarchy tree (GUI state). DroidMate not only allows testers to develop their action selection strategy but offers four default strategies as well. The four strategies offered are random selection, least explored action first, record & playback, and fitness proportionate. Where fitness proportionate uses a mined interaction model to predict the probability of each UI element having an event and uses these probabilities to select the next action to execute. To evaluate the performance of the exploration of the GUI and to monitor for any crashes, DroidMate intercepts all API calls between the Android OS and the AUT. This allows for detailed information to be retrieved as all API calls are intercepted, however, it also means instrumentation of the AUT is required. This implies either the AUT's code must be open source or open to repackaging. If either is not possible DroidMate can not work on the AUT. As DroidMate has inbuilt support for testers to design their action selection approaches, DroidMate is a flexible tool allowing for injecting application-specific information. Overall, their approach does not seem to have unique components that are not present in other scriptless testing tools. Lastly, note that DroidMate does not verify the correctness of the GUI state, no oracle module is present.

PUMA is a systematic exploration tool created by Hao *et al.* focused on analyzing Android applications [55]. Although the main focus of the creators of PUMA is analyzing Android applications, PUMA can be used for scriptless testing without requiring changes to the tool. PUMA consists of two parts, the first part is the exploration component which contains the logic on how the explore the AUT. The default exploration strategy of PUMA is BFS. When residing in a state, all actions at this state are executed and the unique resulting states are added to a stack. Only when the stack is empty PUMA stops exploration.

What defines a unique state is determined by the tester. When a low abstract level is chosen by the tester a difference in small details defines a new unique state, leading to a long exploration stage, but a well-explored application. Vice-versa, a high abstraction level leads to a shorter exploration state, but the application might be less explored. As states are saved on a stack PUMA can explore the AUT in parallel speeding up the process. The second component of PUMA is the analysis part where the logic to analyze the runtime properties of the AUT after each action is located. There is no default implementation for this, the tester needs to specify what it considers non-expected behavior. To help testers specify non-expected behavior, a domain-specific language is created by Hao *et al.* called PUMAScript. If the tester defines no requirements it is assumed analysis of possible erroneous behavior takes place after complete exploration based on the saved state transition graph. Overall, PUMA provides an interesting scriptless testing tool using an exploration strategy that can run in parallel on multiple machines. Puma does not require source code to be available but instead instruments the bytecode of the AUT. Additionally, PUMA has a lot of flexibility as it allows testers to define requirements the AUT needs to adhere to in PUMAScript. The weakness of PUMA also lays in this flexibility. For every application the analysis stage needs to be developed, leading to additional costs for maintenance of PUMA. PUMA also does not make use of an oracle as it does not check the correctness of the state of the GUI. PUMA is an open-source tool available online for all interested parties, however, the codebase has not been maintained since 2014 so PUMA is unlikely to work on new Android versions.

Mirzaei *et al.* describe and implement a scriptless testing tool called SIG-Droid [56]. SIG-Droid uses a symbolic execution approach to achieve scriptless testing for Android applications. In symbolic execution, the AUT is analyzed to determine what inputs cause each conditional branch of the AUT to execute. This results in constraints for each conditional branch present in the AUT. Using these constraints, test cases can be generated. This approach lowers the number of test inputs that have to be tried as the constraints to achieve each possible behavior is known. Usually, this approach is applied to unit tests but SIG-Droid applies it to Android GUIs. SIG-Droid learns the constraints on the conditional branches of the AUT by analyzing the interface model. The behavioral model captures the event-driven behavior of the AUT. Both models are extracted from the source code of the AUT, this indicates SIG-Droid can only work on open source applications. The source code is also used to determine the GUI widgets present on the GUI state. The performance of SIG-Droid is measured through the code coverage it achieves. Overall, it seems like SIG-Droid has a promising approach for unit testing of an application, but the unproven potential for GUI testing. Since SIG-Droid is unavailable, it cannot be inspected or tested on current Android applications. Additionally, SIG-Droid only works on applications where the source code is available, limiting the ability to use the tool on industrial applications and has not oracle component to verify the correctness of a GUI state.

Adamo *et al.* have developed a tool for reinforced learning-based scriptless GUI testing for Android applications [57]. To improve code coverage they propose to use a reinforced learning algorithm called Q-learning [58] for determining what action should be executed at each point in time on the AUT. The concept of Q-learning in the tool proposed by Adamo *et al.* iteratively identifies events that are likely to discover unexplored states and revisit partially explored states. This means that when the Q-learning agent needs to choose an action, each action has a reward based on associated rewards derived from past iterations. The higher the reward for an action, the more likely the Q-learning agent selects this action next. To identify widgets present in the GUI Adamo *et al.* use Appium and the Android UIAutomator. Adamo *et al.* compared the performance of their implementation to UI/Application Exerciser Monkey and noted an improved performance between 3%-18%, depending on the tested application. Overall, Adamo *et al.* focus more on the Q-learning algorithm and the benefit it can potentially provide to the action selector of scriptless testing tools than providing a functioning tool or framework for scriptless testing. As they focus mainly lies on the Q-learning algorithm no oracle component is discussed.

CrashScope is a tool created by Moran *et al.* [59]. CrashScope is designed to achieve systematic exploration-based scriptless GUI testing for Android applications. The unique factor of CrashScope is its ability to generate detailed crash reports when encountering a problem in the AUT. The crash report contains screenshots, reproduction steps, exception trace, and a replayable script to reproduce the crash on the AUT. To find crashes in the AUT CrashScope systematically explores the application using a DFS or random approach. Before trying to apply the DFS or random exploration method to the AUT, static code analysis is used to create a model of the AUT. CrashScope is set up to allow parallel exploration of the AUT, speeding up the testing process. CrashScope's performance is measured by how many reproducible crashes it can locate. In the two empirical evaluations Moran *et al.* claim CrashScope is as effective as the state of the art scriptless testing tools and its crash reports as are useful as human written crash reports. Overall, the unique component of CrashScope is its crash report generation, this indicates CrashScope has an oracle component. In the oracle component designed by Moran *et al.* the focus is on application crashes and the logs generated to determine if the application is in a fault state.

SwiftMonkey[3] is a tool designed by the Zalando company for monkey testing on iOS applications. It essentially is an attempt to create a similar tool as UI/Application Exerciser Monkey but for iOS. The tool cannot be classified as a scriptless testing tool as its only capability is clicking on random screen coordinates of the iOS device. Nevertheless, it is relevant to be aware of the existence of SwiftMonkey as to our knowledge it is the only non scripted testing tool for iOS devices.

2.6 Discussion

In the mobile scriptless testing domain there exist numerous tools focused on the Android system, see Table 2.1 for a summarized overview of the tools. It can be noted DroidBot and DroidMate could be suited for extension. However, DroidMate is limited to the random exploration algorithm as has no other algorithms supported. Its flexibility is mainly to integrate domain specific information, not completely new exploration algorithms. DroidBot has the same problem and is limited in the domain specific information which can be integrated. Additionally, many of the available scriptless testing tools are abandoned, untested on industrial systems, or lack in performance and intelligence. For the iOS operating system there seem to be no scriptless testing tools available. This leaves a promising opportunity to develop an industrial verified Android and iOS scriptless testing tool.

Table 2.1: The table with a summary of the tools presented in Section 2.5. Note the symbols used mean the following; −−:very poor, −:poor, 0:neutral, +:good, ++: very good.

Feature set	**Eskonen et al.**	**Sapienz**	**A3E**	**DroidBot**	**Humanoid**	**Stoat**	**DroidMate**	**PUMA**	**SIG-Droid**	**Adamo et al.**	**CrashScope**
Publicly available	No	No	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
Actively maintained	No	Yes	No	Yes	No	No	Yes	No	No	No	Yes
Exploration algorithm	Reinforced learning	Systematic/ search	Systematic	Systematic, Random	Supervised learning	Stochastic model based	Random	Systematic	Symbolic execution	Reinforced learning	Systematic
Flexibility wrt domain knowledge	−−	+	0	+	−−	0	++	++	−	−	−−
Concept of state	No	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	No	Yes
Failure detection	−−	++	−−	+	−−	+	−−	−	−−	−−	++

From the analysis of the different scriptless testing tool, we extracted a number of design aspects which can be used to create a better performing, industrial tested, scriptless testing application (see Section 2.6.1).

[3]`https://github.com/zalando/SwiftMonkey`

For automated testing there is research work being conducted to improve oracles used in testing. However, for scriptless testing in the mobile domain not all automated testing oracles are suitable and at the time of writing there seems to be very limited oracle research available for mobile scriptless testing tools. The only oracles available seem to be log and crash oracles. This leaves an opportunity to work on oracles for scriptless testing in the mobile environment.

2.6.1 Scriptless Design Aspects

From the scriptless testing approaches and tools highlighted in section 2.5 we extracted a number of design aspects important for scriptless testing. These design aspects can be leveraged in the mobile scriptless GUI testing tool to be implemented in this thesis. The following design aspects have been highlighted in the related work which we believe to have value.

- Exploration algorithm
- Flexibility to add domain specific AUT information
- GUI state information
- Concept of state
- Failure detection

The exploration algorithm is a core design aspect as it determines which actions are contained in the test sequences. With the wrong exploration algorithm, intelligence in creating the test sequences can be missing, thus a larger part of the application may stay uncovered compared to a smarter exploration algorithm. Therefore exploration algorithms are essential for a scriptless testing tool. For exploration algorithms to work well memory is required to log what components of the application have been explored and which actions have been taken, a state model is used for this. Therefore, there must be a concept of state and a state model in the scriptless testing tool. Additionally, the concept of state allows for abstraction such that unnecessary details can be skipped over in test sequence generation. The flexibility to add domain-specific AUT information to the scriptless testing tool is an important design aspect as one would like to keep this as simple as possible for the tester. Making this easier allows for better maintenance of the domain-specific information. To work with a GUI it is important to understand the GUI. If there are more details available on the GUI state, more information can be leveraged for creating test sequences. Therefore, the ability to gain GUI state information and the ability to gain detailed information is important. Lastly, the goal of scriptless testing is to find failures in the AUT. This means the detection of failure is important for the functioning of a scriptless testing tool, leading to failure detection being an important design aspect of a scriptless testing tool.

Chapter 3

Mobile scriptless GUI testing

The goal of this thesis is to develop a scriptless testing tool for mobile apps (Android and iOS [60]) and apply the tool to an industrial use case. In Chapter 2, a number of existing scriptless testing tools for the Android operating system are introduced. However, these tools have shortcomings. All of the publicly available tools are academic tools, often lacking maintenance. The tools which are still maintained are often not verified on industrial-sized mobile applications or lack intelligent exploration/ test sequence generation algorithms. Additionally, all available tools lack a failure detection component (test oracle). Despite these shortcomings, design aspects can be learned from the existing tools. In this Chapter, we extend the previously introduced scriptless testing tool TESTAR for Android and iOS applications, we name the extended TESTAR tool mTESTAR. Ideally, we create a tool with a shared core for Android and iOS scriptless testing to minimize maintenance costs. mTESTAR will be validated on the Android ING Bankieren mobile application[1], in order to ensure that its design and implementation are correct (see Chapter 4).

In this chapter, the components relevant to the design and implementation of the mTESTAR tool are described. The first section described the architecture of mTESTAR as well as the changes required to make the mTESTAR architecture suited for mobile applications, see Section 3.1. Followed by a section focusing on the new accessibility API integrated into mTESTAR to allow for interaction with mobile applications, see Section 3.2. Next, the two modes of mTESTAR relevant for mobile scriptless testing are described, namely, Spy mode, see Section 3.3, and Generate mode, see Section 3.4. The generate mode section consists of sub-sections discussing the different components relevant to achieve the generation of test sequences. As highlighted previously in Section 2, the existing tools lack failure detecting capabilities and scriptless testing oracles are under-explored. Therefore, we introduce the concept of test oracles for scriptless testing and how it can be leveraged in the context of scriptless mobile testing, see Section 3.5.

3.1 Architecture of mTESTAR

Figures 3.1 and 3.2 show the mTESTAR architecture. In Figure 3.1, the mTESTAR components and their relations are displayed. Components that are modified (compared to TESTAR) for scriptless testing of mobile applications are colored orange. Before discussing the architecture in more detail, it is important to note that the general execution flow of mTESTAR cannot be observed through these two figures. The execution flow of mTESTAR is the same as TESTAR's execution flow, which can be observed in Figure 2.1.
The accessibility API allows mTESTAR to retrieve the current structure and component information of the AUT. At the same time it also allows mTESTAR to execute actions on the host platform of the

[1] `https://play.google.com/store/apps/details?id=com.ing.mobile`

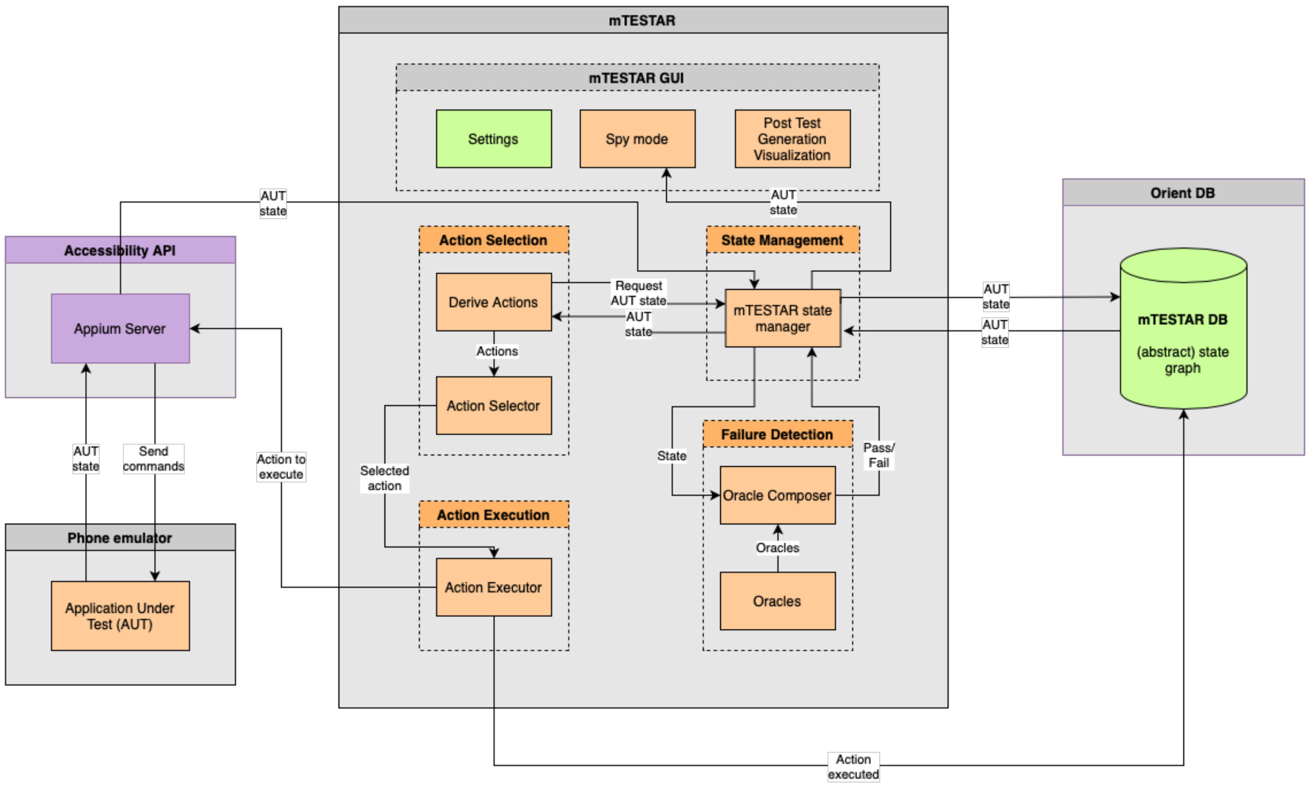


Figure 3.1: The architecture of the implementation of mTESTAR for mobile applications. Components colored in orange have been modified for mTESTAR. Purple components indicate an external service. Green components are unmodified.

AUT. It can be observed that the accessibility API, Appium in our case, is located between the mobile AUT and mTESTAR, more detail on how the mobile-specific accessibility API Appium is leveraged in Section 3.2. The OrientDB is an open-source graph database used as the memory of mTESTAR. It is used to store detected states, and actions taken by mTESTAR.

3.1.1 State Management

The State Management component is the central component of mTESTAR which takes care of capturing the GUI state of the AUT, saving the GUI state to the OrientDB, and managing the abstract state and abstract actions, see Figure 3.2. The concept of state is important to manage in a scriptless testing tool as it is required for certain exploration algorithms (e.g. Q-learning, unvisited actions first). When the State Management component receives the GUI state from the accessibility API, it translates the GUI state into an internal mTESTAR state. Translating the GUI state for the AUT already existed in TESTAR for web and desktop applications. However, Appium is not the same accessibility API as in either web or desktop TESTAR, therefore the engine which builds the mTESTAR state from the received GUI state has been adapted. The GUI state that Appium passes is an XML document with the different Android or iOS widgets in a tree format. This tree is parsed and each element is saved as a widget together with its corresponding attributes. Although Appium claims the data it provides abstracts away from the OS from which the information is obtained, in practice there are significant differences between iOS and Android. For example, Appium obtains different widget attributes for Android and IOS applications. Due to this difference, two engines have been constructed generating mTESTAR state for Android and iOS separately. When saving the state to the OrientDB, the obtained mTESTAR state is combined with

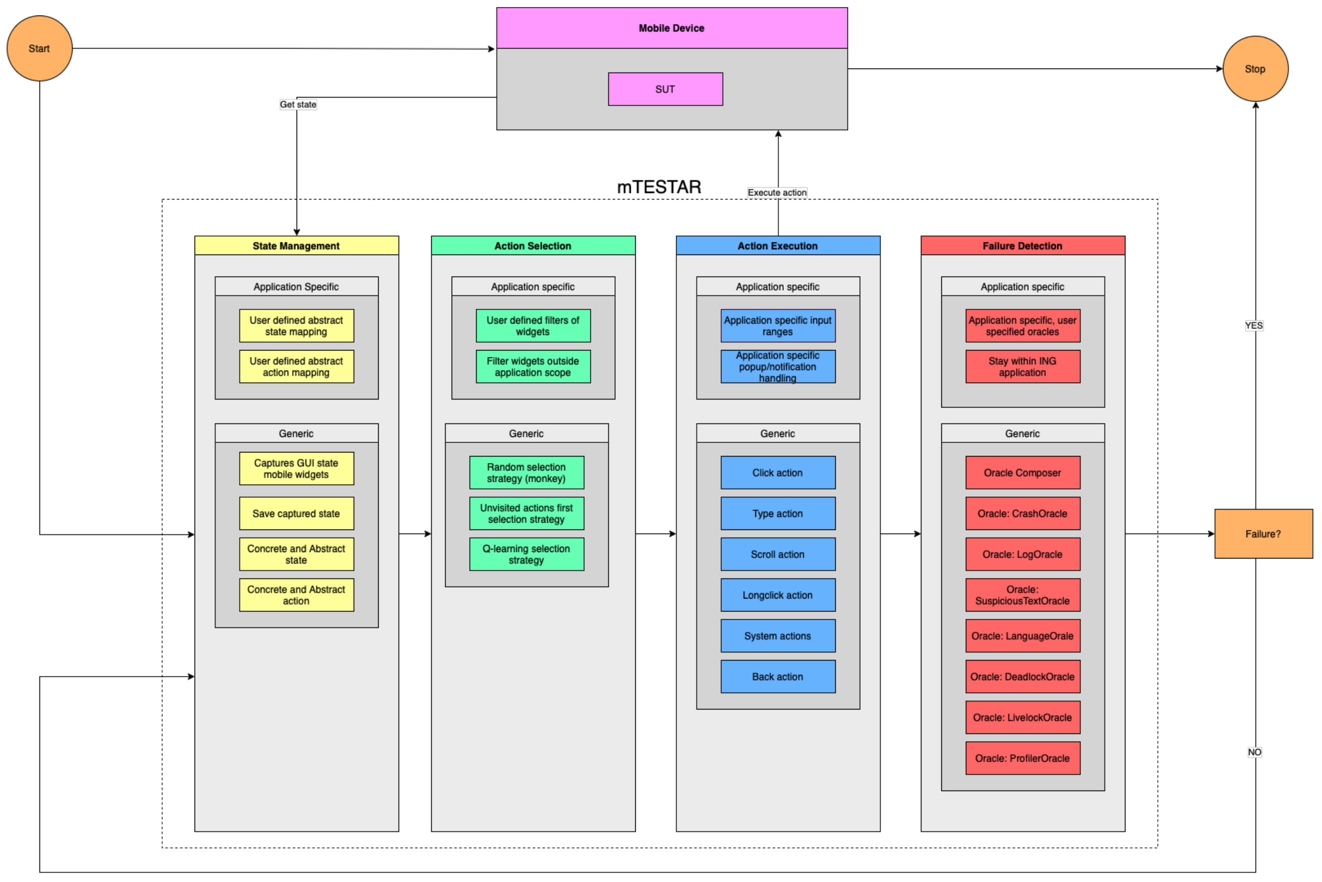

Start
Mobile Device
SUT
Stop
Get state
mTESTAR
Execute action
State Management
Application Specific
User defined abstract state mapping
User defined abstract action mapping
Generic
Captures GUI state mobile widgets
Save captured state
Concrete and Abstract state
Concrete and Abstract action
Action Selection
Application specific
User defined filters of widgets
Filter widgets outside application scope
Generic
Random selection strategy (monkey)
Unvisited actions first selection strategy
Q-learning selection strategy
Action Execution
Application specific
Application specific input ranges
Application specific popup/notification handling
Generic
Click action
Type action
Scroll action
Longclick action
System actions
Back action
Failure Detection
Application specific
Application specific, user specified oracles
Stay within ING application
Generic
Oracle Composer
Oracle: CrashOracle
Oracle: LogOracle
Oracle: SuspiciousTextOracle
Oracle: LanguageOrale
Oracle: DeadlockOracle
Oracle: LivelockOracle
Oracle: ProfilerOracle
YES
Failure?
NO

Figure 3.2: More detailed information on the sub-components of the mTESTAR architecture shown in Figure 3.1. Note that this figure abstracts away from the external accessibility API and the connection between the different components.

a screenshot. We have changed the procedure to obtain screenshots through the accessibility API. The process of saving the mTESTAR state and screenshot to the OrientDB has been left unchanged. Abstract state and abstract action are important concepts for the performance of the test sequence generation, therefore we will discuss then in Section 3.4.2 and 3.4.3 respectively.

3.1.2 Action Selection

The Action Selection component of mTESTAR takes care of deriving the possible actions which can be taken on a specific mTESTAR state. Consequently, one action is selected from the list of possible actions, and executed. The action derive components are newly build for both Android and iOS, as deriving the possible actions depends on the widgets and their attributes. For Android, the actions *click*, *long-click*, *scroll*, *type*, *back*, and *system* actions have been implemented. For iOS *click*, *scroll*, and *type* actions have been implemented. *Long-clicks* and *back* actions are not implemented for iOS given that these functions do not exist. Secondly, *system* actions are not supported for iOS due to their absence in Appium. To determine which of the implemented actions are possible on which widgets of the state, each widget in the state is iterated over and its attributes are inspected. When predefined widget attribute values are found, the action and the widget on which the action can be executed are added to the derived actions list. E.g. Android widgets carry the property *clickable*, if this boolean is true we add the widget with the click action to the derived actions list. Unfortunately, not all actions have clear mappings between widget attributes and actions. In these cases the widget class property can be used. E.g. iOS class attribute being *XCUIElementTypeButton* means the widget is clickable. Which classes map to each action depends on the AUT.

Once a list of possible actions has been derived for a state, it must be determined what action will be executed. mTESTAR has support for three different action selection algorithms; 1) *Random* where an action is chosen from the derived list at random, 2) *Unvisited actions first* where the state model in the OrientDB is leveraged to select an action that has not previously been executed, and 3) the reinforced learning algorithm *Q-learning* [44]. The action selection algorithm returns the action to be executed, which is passed to the action execution component. Note that action selection algorithms are out of the scope of this thesis, therefore we have not improved upon the action selection algorithms currently present in TESTAR. For an overview of the action selection algorithms/ GUI exploration algorithms see Section 2.2.

3.1.3 Action Execution

The Action Execution component is dedicated to getting the selected action executed on the AUT. As mentioned before, Android supports the actions *click*, *long-click*, *scroll*, *type*, *back*, and *system* actions, while iOS supports *click*, *scroll*, and *type* actions. The Action Execution component takes the action which should be executed and translates it to Appium commands representing the same action. In turn, Appium is instructed to execute the passed command on the AUT and return the results back to the Action Execution component. Additionally, the newly received AUT state is passed to the State Management component. The State Management component is required to be informed of new states as it is the component in charge of distributing the GUI state to other components of mTESTAR.

The last core component of mTESTAR is failure detection, also called the oracle component. Failure recognition is a critical element in mTESTAR and required for a well functioning mobile scriptless testing application. Additionally, the oracle component contains several additions for mTESTAR, therefore we will discuss it in detail in Section 3.5.

3.1.4 mTESTAR GUI

The mTESTAR GUI is used for multiple purposes. The mTESTAR GUI enables the tester to configure mTESTAR at runtime. Next, testers can use the spy mode to get additional information about the AUT widgets visually, see Section 3.3. Lastly, testers can view the test sequences generated once mTESTAR has finished. As a mobile operating system significantly differs from web and desktop operating systems, we have decided to no longer use the OS of the device which runs mTESTAR for visualization. Therefore, both the spy mode and the post test sequence generation visualization have been modified. Note that the settings (E.g. number of sequences, number of actions, etc.) which can be adjusted in the mTESTAR GUI have not been modified in this thesis.

The Post Test Generation Visualization component visualizes the sequence of actions taken by mTESTAR in the generate mode. This can be used by the tester to inspect if a crash occurs on the application being tested. Using the Post Test Generation Visualization it is possible to visually retrace the test sequences mTESTAR has generated to produce a crash. To provide this functionality, mTESTAR creates an HTML file with information on each state, the action highlighted, and detailed information about the widget on which the action takes place.

3.1.5 Design aspects

It is important that the design aspects deemed important from the literature research, see Section 2.6.1, are incorporated into the different architectural components of Figure 3.1. The important design aspects mentioned were; exploration algorithms, flexibility to add domain-specific AUT information, GUI state information, the concept of state, and failure detection. Each of these design aspects are present in one or more components of mTESTAR. The exploration algorithms are incorporated in the action selection component, they are used to determine which action is executed on the AUT. Detailed GUI state information is obtained from the AUT through the accessibility API, and the state management component takes care of translating it to an mTESTAR state. The state management component also ensures the concept of state is present in mTESTAR and that state abstraction can be used. The oracles of mTESTAR ensure failure detection is incorporated. Lastly, mTESTAR supports the flexibility to add domain-specific AUT information throughout all its architectural components. State management allows testers to determine the abstraction for states and actions to be used throughout the testing. In the oracle composer, testers can define their composition and individual oracles. While the action selection component ensures testers can modify the action derivation algorithm.

3.2 Accessibility API

mTESTAR obtains information and executes actions on the GUI of AUT's through the accessibility API [42], no explicit environment information is used in mTESTAR outside of the accessibility API. A framework that is designed for automating GUI tests on mobile applications, specifically the Android and iOS platform, is Appium [61]. The reason for choosing Appium as accessibility API is threefold.

Firstly, Appium returns the structure of the GUI of the AUT on the widget level, allowing mTESTAR access to information on the individual components of the AUT. This aids with understanding the GUI opposed to working with, for example, screenshots.

Secondly, Appium provides a single API for both Android and iOS such that developers do not need to maintain two different setups for two API's separately[2]. It interacts with Android applications through Google's UiAutomator2[3] and with iOS applications through Apple's XCUITest[4].

Thirdly, Appium can be added to the application at runtime. By using the vendor-provided automation frameworks UIAutomator2 and XCUITest, Appium makes sure that compiling specific Appium or third-party code into the application under tests is not necessary, this allows the tester to be sure that the application tested is the same as the application that is being released.

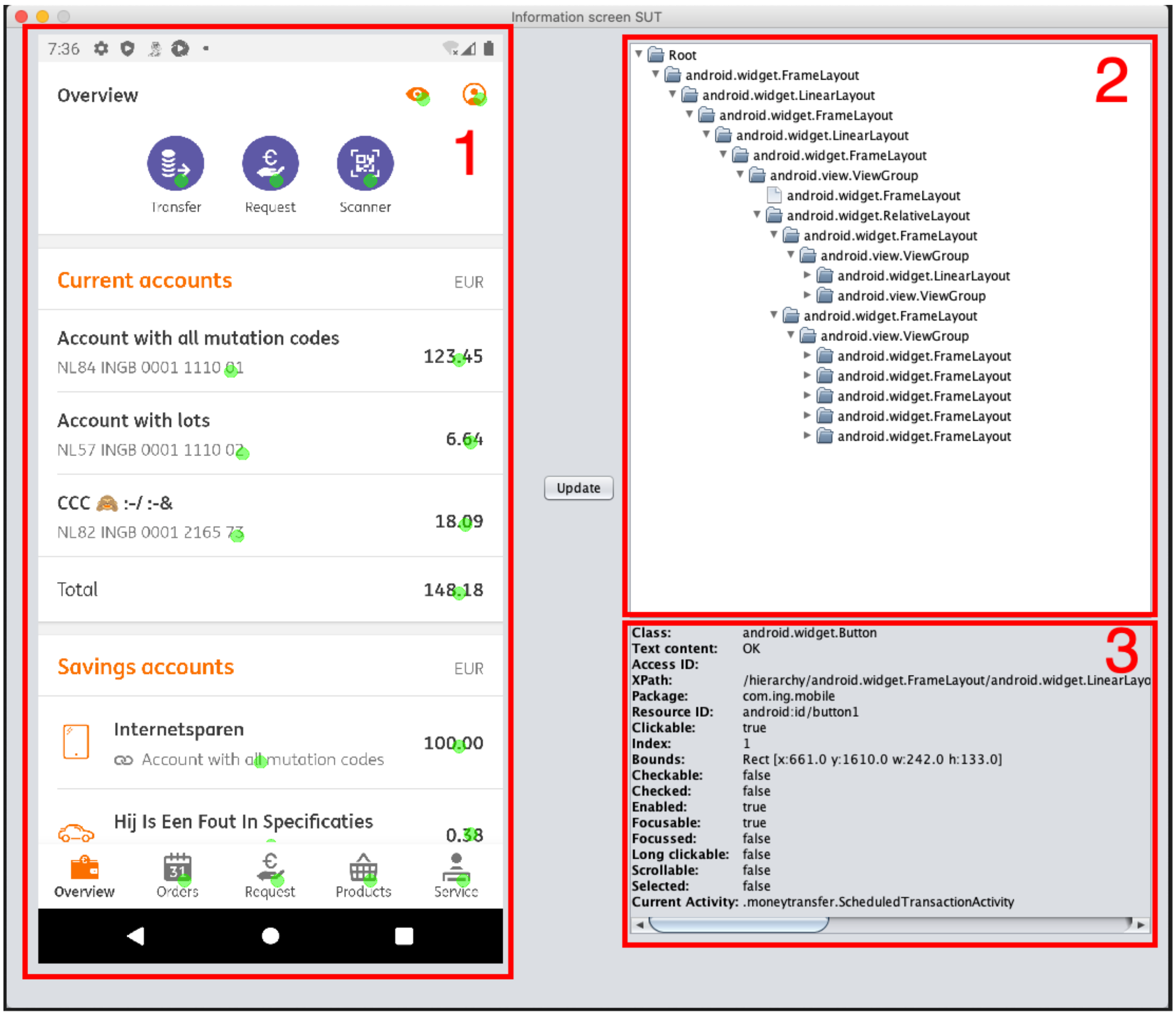
Appium works with the concept of a session; a connection between the AUT and the Appium server is set up and a connection between the Appium server and mTESTAR is set up. Appium continuously listens to mTESTAR to see if any commands are being sent, if any commands are received it translates these commands and pushes them to the mobile device. The accessibility operations implemented in mTESTAR are actions, system actions, screenshots, obtaining GUI state, obtaining AUT states, and starting and stopping applications. As the Appium implementation of these operations differs between Android and iOS, two separate Appium implementations are created in mTESTAR.

[2]https://appium.io/docs/en/about-appium/intro
[3]https://developer.android.com/training/testing/ui-automator
[4]https://developer.apple.com/library/archive/documentation/DeveloperTools/Conceptual/testing_with_xcode/chapters/01-introduction.html#//apple_ref/doc/uid/TP40014132-CH1-SW1

It is important to note the difference in performance for Appium between Android and iOS. Executing operations on the AUT does not delay mTESTAR in either of the operating systems. However, retrieving the state takes significantly longer for iOS as Appium in iOS retrieves all widgets of the GUI state, not only the visible ones. This leads to a massive performance hit. For Android retrieving the GUI state takes on average 10ms on the ING application we tested for. For iOS, it takes around 2000ms to retrieve the GUI state on the ING application under tests. This significantly slows down mTESTAR execution for iOS applications as obtaining the GUI state is common action within mTESTAR. The poor performance for the GUI state retrieval for iOS is a point of concern which needs attention in future work.

3.3 Spy mode in mTESTAR

The Spy mode of mTESTAR is an inspection mode where the GUI of the application under test can be examined in detail. Besides showing the visible components, the Spy mode provides attribute information on each widgets of the application. The widget information can be used for setting up the test sequence generation of mTESTAR. If the tester requires no additional information, the spy mode can function as an exploration tool. For example, during exploration the tester can determine what the right settings for mTESTAR's sequence length and the number of sequences are.

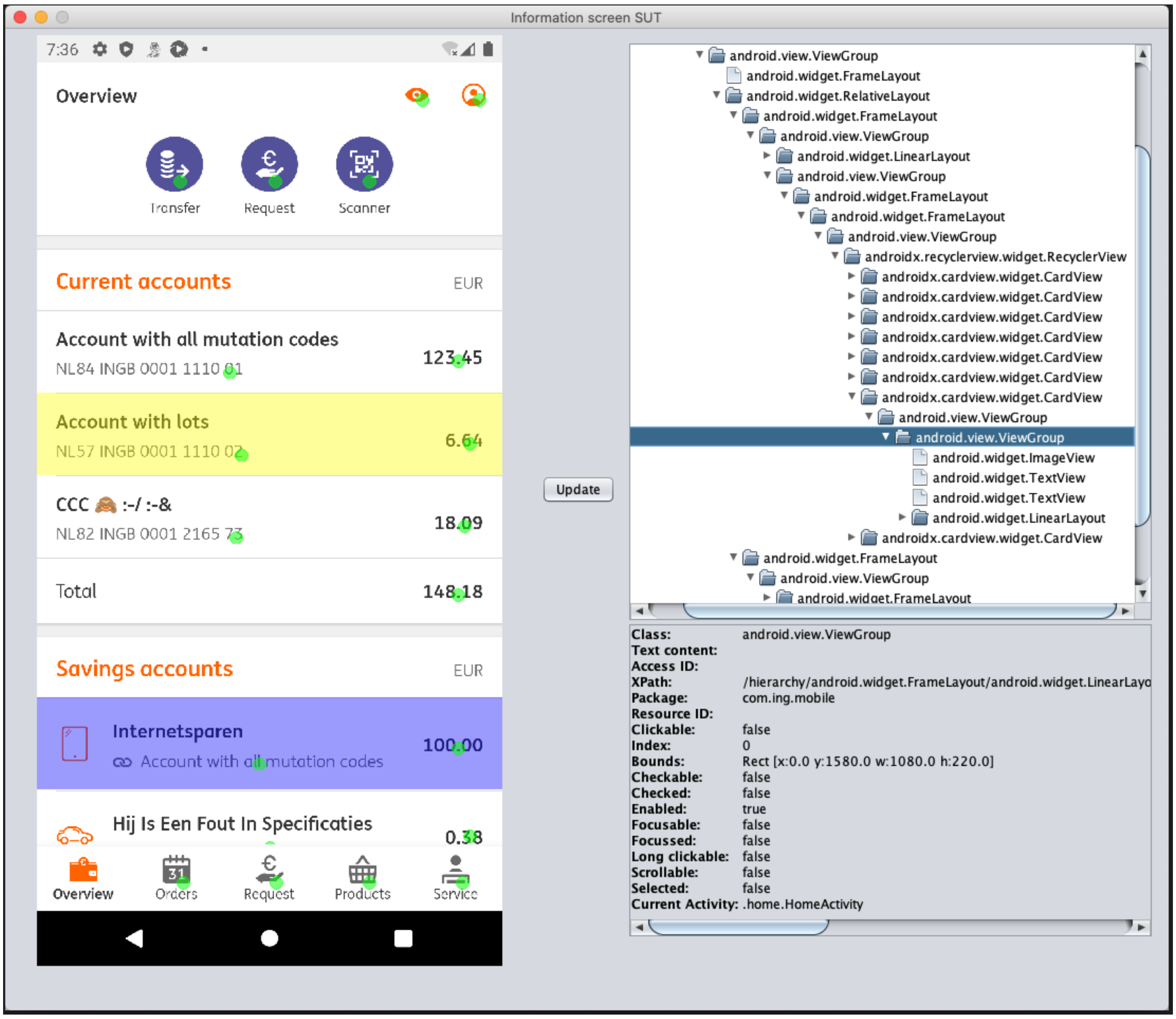


```
▼ Root
  ▼ android.widget.FrameLayout
    ▼ android.widget.LinearLayout
      ▼ android.widget.FrameLayout
        ▼ android.widget.LinearLayout
          ▼ android.widget.FrameLayout
            ▼ android.view.ViewGroup
                android.widget.FrameLayout
              ▼ android.widget.RelativeLayout
                ▼ android.widget.FrameLayout
                  ▼ android.view.ViewGroup
                    ▶ android.widget.LinearLayout
                    ▶ android.view.ViewGroup
                ▼ android.widget.FrameLayout
                  ▼ android.view.ViewGroup
                    ▶ android.widget.FrameLayout
                    ▶ android.widget.FrameLayout
                    ▶ android.widget.FrameLayout
                    ▶ android.widget.FrameLayout
                    ▶ android.widget.FrameLayout
```

Class:	android.widget.Button
Text content:	OK
Access ID:	
XPath:	/hierarchy/android.widget.FrameLayout/android.widget.LinearLayo
Package:	com.ing.mobile
Resource ID:	android:id/button1
Clickable:	true
Index:	1
Bounds:	Rect [x:661.0 y:1610.0 w:242.0 h:133.0]
Checkable:	false
Checked:	false
Enabled:	true
Focusable:	true
Focussed:	false
Long clickable:	false
Scrollable:	false
Selected:	false
Current Activity:	.moneytransfer.ScheduledTransactionActivity

Figure 3.3: Screenshot of Spy mode of the home page after login for the ING android application.

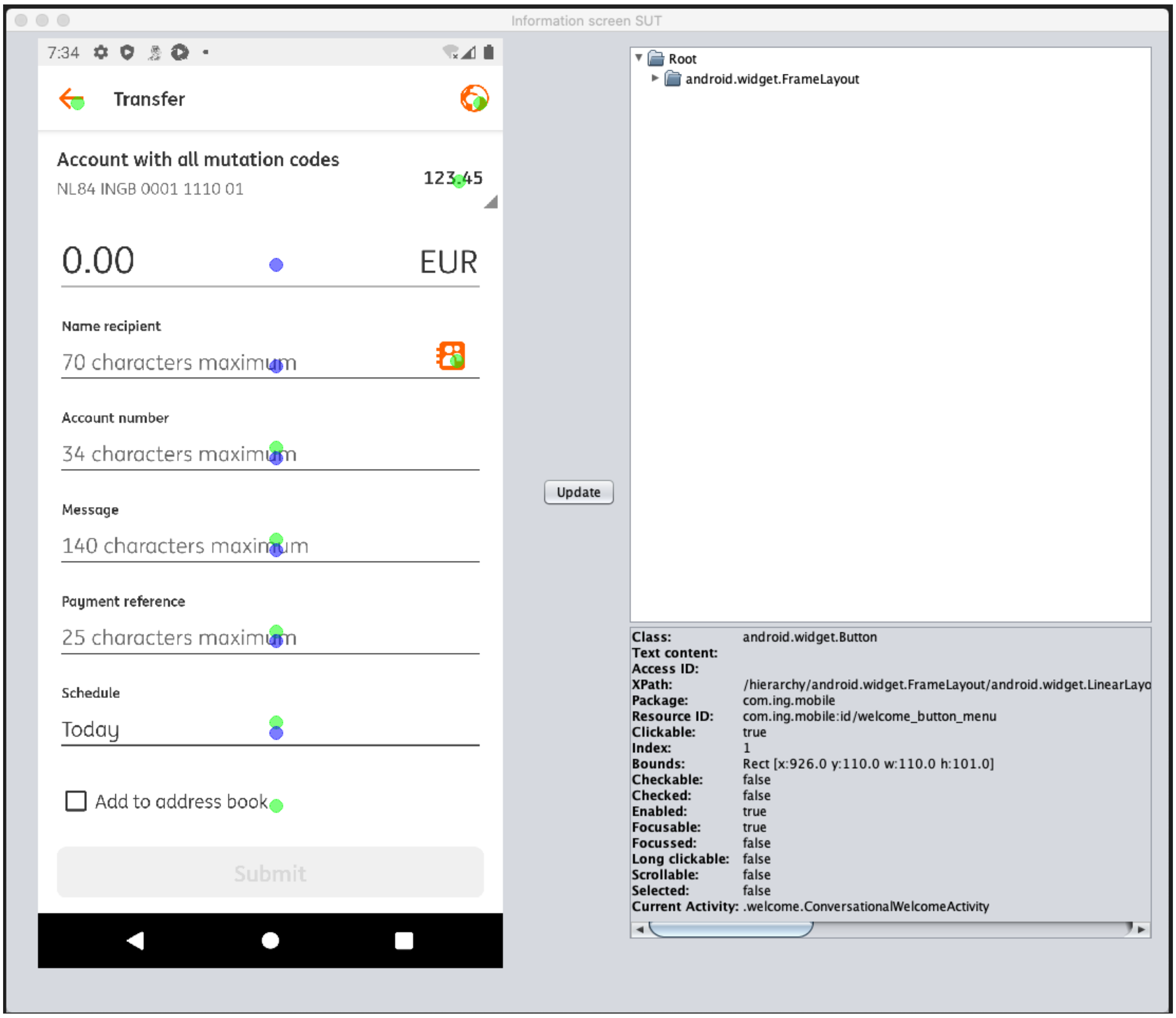
In Figure 3.3 the spy mode running on the Android ING application can be observed. Box number one shows a screenshot of the mobile device on which the ING application is running. Additionally, widgets on which actions can be taken are highlighted with colored dots. In the case of Figure 3.3, the green dots indicate click actions (note that in Figure 3.5 another type of action highlight can be observed, namely blue dots highlighting widgets where type actions are possible). Box number two shows the hierarchical tree of the elements present in the current GUI state of the AUT. The elements in the tree can be clicked and will be highlighted on the screenshot and vice versa, see Figure 3.4. In box number three additional information of the selected widget can be observed, all attribute information known to mTESTAR about the selected widget is displayed here.



```
▼ android.view.ViewGroup
    android.widget.FrameLayout
  ▼ android.widget.RelativeLayout
    ▼ android.widget.FrameLayout
      ▼ android.view.ViewGroup
        ▶ android.widget.LinearLayout
        ▼ android.view.ViewGroup
          ▼ android.widget.FrameLayout
            ▼ android.widget.FrameLayout
              ▼ android.view.ViewGroup
                ▼ androidx.recyclerview.widget.RecyclerView
                  ▶ androidx.cardview.widget.CardView
                  ▶ androidx.cardview.widget.CardView
                  ▶ androidx.cardview.widget.CardView
                  ▶ androidx.cardview.widget.CardView
                  ▶ androidx.cardview.widget.CardView
                  ▶ androidx.cardview.widget.CardView
                  ▼ androidx.cardview.widget.CardView
                    ▼ android.view.ViewGroup
                      ▼ android.view.ViewGroup
                          android.widget.ImageView
                          android.widget.TextView
                          android.widget.TextView
                        ▶ android.widget.LinearLayout
                  ▶ androidx.cardview.widget.CardView
    ▼ android.widget.FrameLayout
      ▼ android.view.ViewGroup
        ▶ android.widget.FrameLayout
```

```
Class:             android.view.ViewGroup
Text content:
Access ID:
XPath:             /hierarchy/android.widget.FrameLayout/android.widget.LinearLayo
Package:           com.ing.mobile
Resource ID:
Clickable:         false
Index:             0
Bounds:            Rect [x:0.0 y:1580.0 w:1080.0 h:220.0]
Checkable:         false
Checked:           false
Enabled:           true
Focusable:         false
Focussed:          false
Long clickable:    false
Scrollable:        false
Selected:          false
Current Activity:  .home.HomeActivity
```

Figure 3.4: A screenshot of the Spy mode where an element of the tree is clicked and another widget is hovered over in the screenshot of the ING application.



```
Class:             android.widget.Button
Text content:
Access ID:
XPath:             /hierarchy/android.widget.FrameLayout/android.widget.LinearLayo
Package:           com.ing.mobile
Resource ID:       com.ing.mobile:id/welcome_button_menu
Clickable:         true
Index:             1
Bounds:            Rect [x:926.0 y:110.0 w:110.0 h:101.0]
Checkable:         false
Checked:           false
Enabled:           true
Focusable:         true
Focussed:          false
Long clickable:    false
Scrollable:        false
Selected:          false
Current Activity:  .welcome.ConversationalWelcomeActivity
```

Figure 3.5: A screenshot of Spy mode for a screen with click and type actions for the ING android application.

An important design choice worth highlighting is the synchronization of the Spy mode with the actual GUI state of the AUT. There are three possible approaches for synchronization of the Spy mode interface; synchronization on command, synchronization on change, continuous synchronization. In synchronization on command the user determines when the Spy mode should retrieve new information, in synchronization on change the Spy mode interface is only updated when changes are detected, and in continuous synchronization the Spy mode interface continuously pulls the AUT GUI state and refreshes its information. In Table 3.1 the pros and cons for each approach are listed.

Table 3.1: The table which shows the pros and cons for each spy mode synchronization approach.

	Pro's	**Con's**
Synchronization on command	**Performance:** No overhead of unnecessary synchronizations or state comparisons. **Reliability:** no comparisons of state are being computed to determine if changes occurred, thus no wrong comparisons results can be reached. **Control:** user is in control with determining when the Spy mode should refresh.	**Delay:** When changes occur at the AUT GUI the changes are not reflected to the Spy mode until the user explicitly takes an action to retrieve the updates. **Non-consistency:** the Spy mode for other platforms (web, desktop) does not work with synchronization on command. Thus, using it for mobile would cause inconsistency between platforms within (m)TESTAR.
Synchronization on change	**Consistency:** AUT GUI and the Spy mode stay consistent. If the AUT GUI changes, the Spy mode visualization will also change. **Balance performance vs consistency:** As the Spy mode is only updated when changes occur, resources are saved. The comparison of GUI states can happen in the background.	**State difference:** Need to correctly compute whether the GUI state differs from the Spy mode state without requiring to much computation time. **Overhead:** Additional state comparison overhead, but as this can happen separately from the Spy mode GUI thread, impact is limited.
Continuous synchronization	**Consistency:** any changes of the AUT GUI are reflected in the Spy mode. **Overhead:** No state comparison overhead.	**Overhead:** very high GUI overhead as GUI elements are continuously redrawn in the Spy mode.

After evaluating the pros and cons of the three synchronization approaches we have opted for the synchronization on change. This ensures the Spy mode is synchronized with the AUT GUI but does not negatively impact the performance as the GUI elements are only redrawn when a change is detected. The state comparison algorithm implemented compares the widgets and widget attributes and thus should detect any changes to the AUT GUI.

3.3.1 Implementation spy mode

In Figure 3.6 the class diagram of the implementation of the Android spy mode can be seen, the iOS spy mode class diagram is identical except for object names. Four main classes can be observed in the diagram, the general class *AndroidVisualization* which passes information between the components visible in the screenshot. The class *OverlayVisualization*, which supports the highlighting of widgets on the screenshots by plotting *OverlayBox* elements. Lastly, the *TreeVisualization* class takes care of constructing the hierarchical tree from the mTESTAR state and displays the widget information when a specific element is clicked.

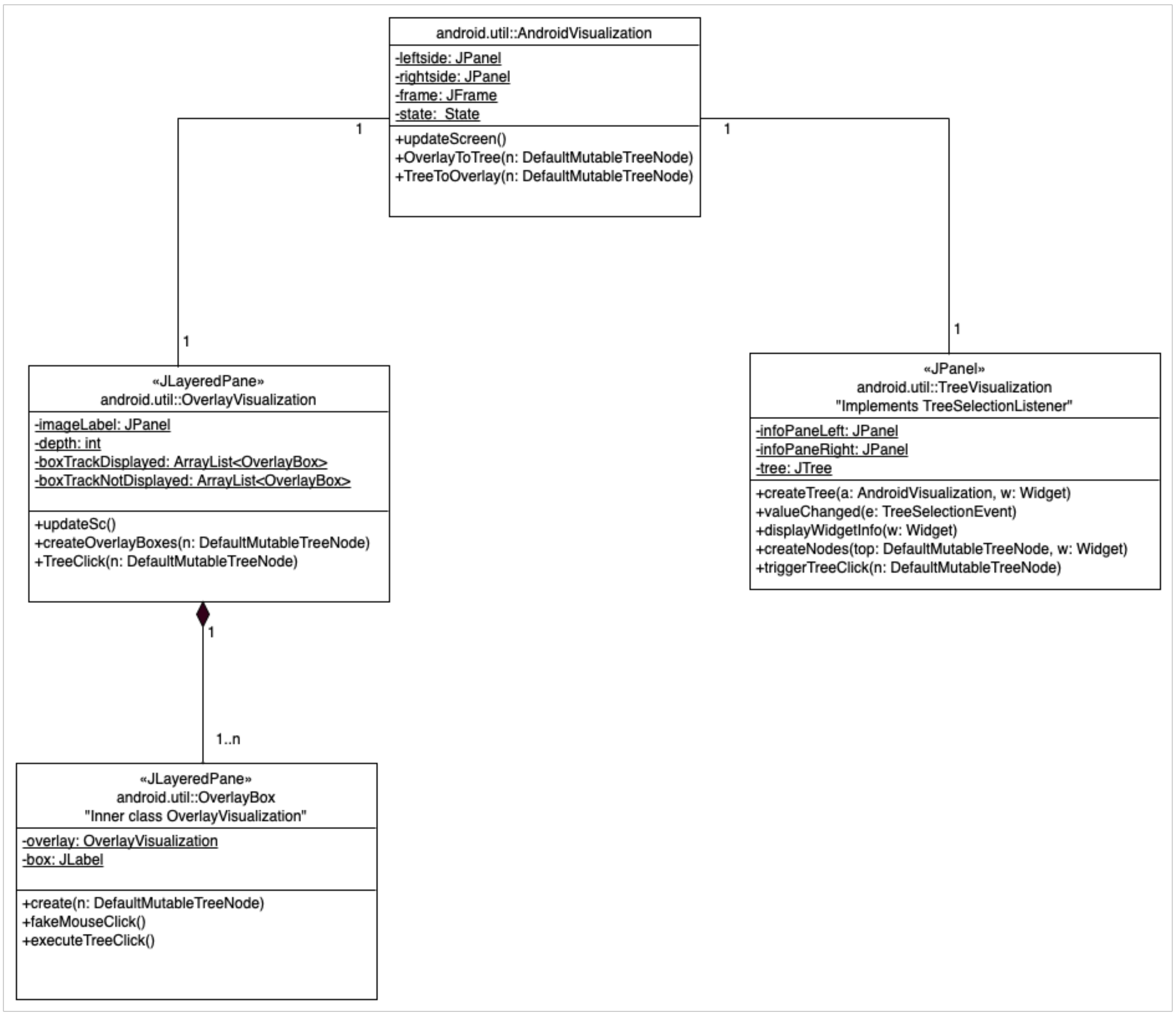

android.util::AndroidVisualization
-leftside: JPanel
-rightside: JPanel
-frame: JFrame
-state: State
+updateScreen()
+OverlayToTree(n: DefaultMutableTreeNode)
+TreeToOverlay(n: DefaultMutableTreeNode)
1
1
1
1
«JLayeredPane»
android.util::OverlayVisualization
-imageLabel: JPanel
-depth: int
-boxTrackDisplayed: ArrayList<OverlayBox>
-boxTrackNotDisplayed: ArrayList<OverlayBox>
+updateSc()
+createOverlayBoxes(n: DefaultMutableTreeNode)
+TreeClick(n: DefaultMutableTreeNode)
«JPanel»
android.util::TreeVisualization
"Implements TreeSelectionListener"
-infoPaneLeft: JPanel
-infoPaneRight: JPanel
-tree: JTree
+createTree(a: AndroidVisualization, w: Widget)
+valueChanged(e: TreeSelectionEvent)
+displayWidgetInfo(w: Widget)
+createNodes(top: DefaultMutableTreeNode, w: Widget)
+triggerTreeClick(n: DefaultMutableTreeNode)
1
1..n
«JLayeredPane»
android.util::OverlayBox
"Inner class OverlayVisualization"
-overlay: OverlayVisualization
-box: JLabel
+create(n: DefaultMutableTreeNode)
+fakeMouseClick()
+executeTreeClick()

Figure 3.6: The class diagram of the Spy mode implementation for Android and iOS applications.

3.4 Generate mode in mTESTAR

The Generate mode is the main focus of mTESTAR. In this mode, the GUI is explored through several sequences consisting of various actions. More mistakes within the AUT are detected when the sequences have a greater coverage of the AUT. Several design aspects play an important role for the generate mode to function correctly and efficiently. In this section, we will discuss the state model, the concept of abstract state, and the concept of abstract action. One can note that the action selection algorithms have a large effect on GUI exploration. However, the concept of action selection algorithms falls outside the scope of this thesis. The action selection mTESTAR does support are; random action selection, unvisited actions first, and Q-learning.

3.4.1 State model in mTESTAR

The mTESTAR state and state model have already been introduced in Section 2.4.2. The concept of state and the state model are very important for the *Q-learning* and *unvisited actions first* action selectors as they need to be aware of which actions and states have already been discovered. As GUIs have a large volume of output [62], storing and dealing with every small change in the GUI as a separate state becomes infeasible and can limit the effectiveness of mTESTAR in exploring the application. To solve this state-explosion problem [45], we can abstract away from certain details in a state, this leads to multiple similar GUI states being grouped, a group is also called an abstract state. See Figure 3.7 and Figure 3.8 for an illustration of how this helps with preventing unnecessarily repeated action execution.

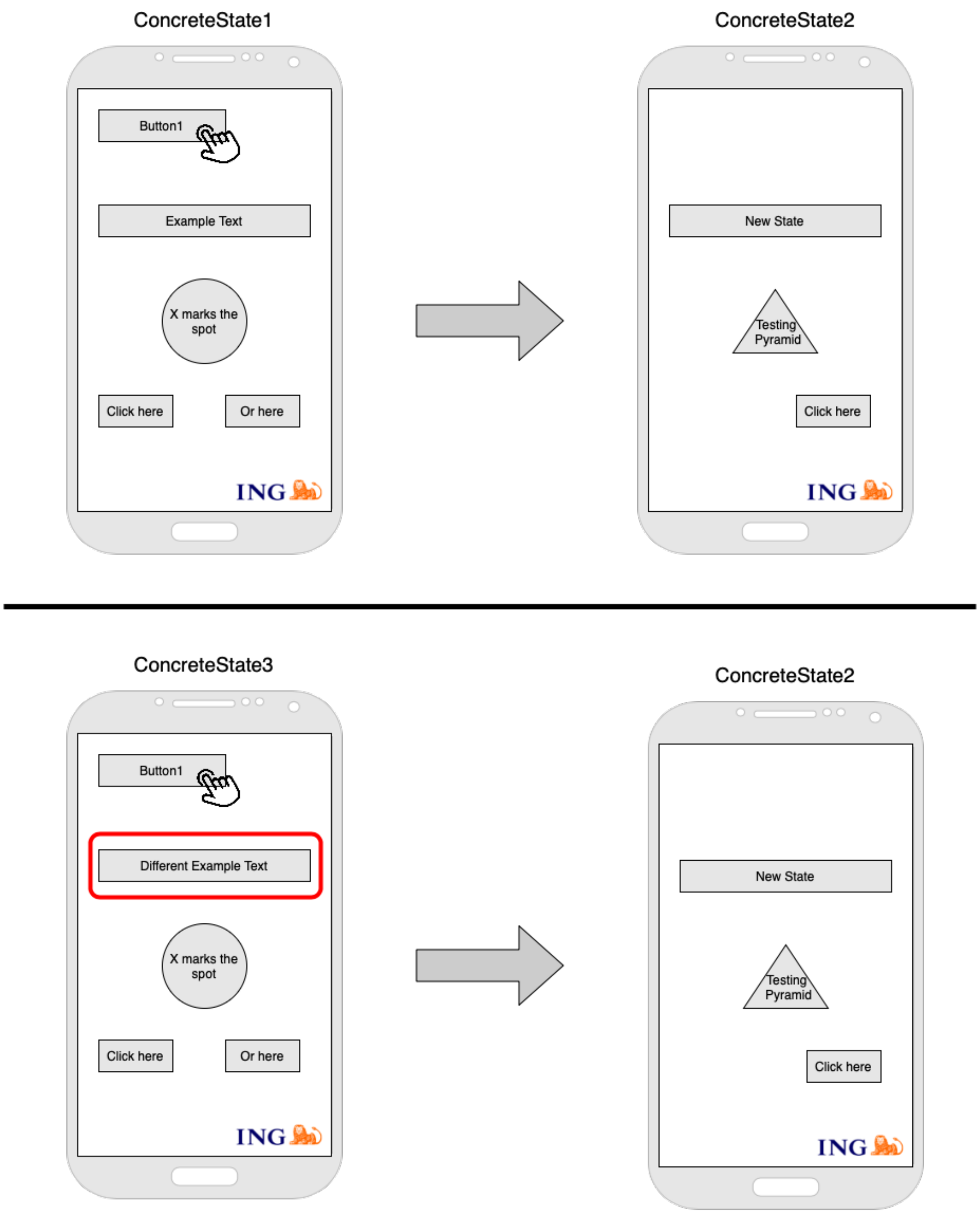


Figure 3.7: Figure illustrating button click effect when using concrete state.

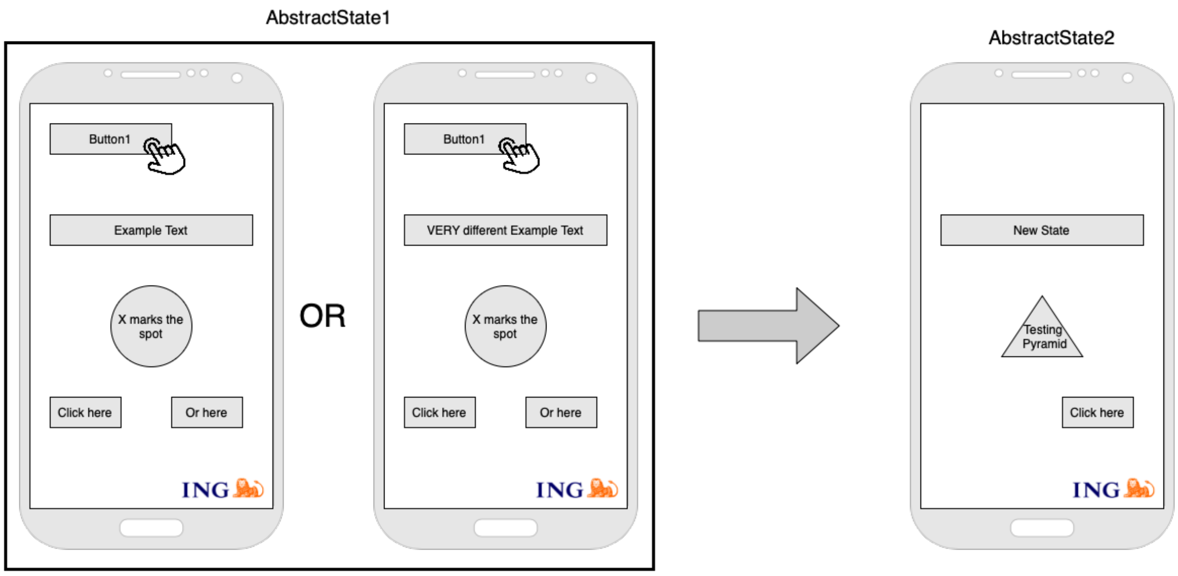


Figure 3.8: Figure illustrating button click effect when using abstract state.

In Figure 3.7 *ConcreteState1* and *ConcreteState3* are very similar, the only difference being the text in the textbox. For both states clicking *Button1* leads to *ConcreteState2*. However, as the state model sees *ConcreteState1* and *ConcreteState3* as different states it interprets clicking *Button1* in each state as a completely different action. In practice both states are almost identical and the resulting state is the same. Therefore, executing both actions is very likely not to find any new faults compared to executing one of the actions. In Figure 3.8 we defined the abstract state in such a way that *ConcreteState1* and *ConcreteState3* now are recognized as *AbstractState1* (see Section 3.4.2). Now when Button1 is clicked in either concrete state it will not be recognized as a new action in the other concrete state, this results in less repeated actions and thus requiring less runtime to achieve coverage.

3.4.2 Abstract state

Mobile applications GUIs are very different from web or desktop applications due to the smaller screen of mobile phones [63]. Thus, the abstract state definition for mobile applications will significantly differ from desktop or web applications. The abstract state definition must be defined for every application under test separately, to get optimal performance out of mTESTAR and its state model. In the case of mobile applications we define which widget attributes should be compared to differentiate states. When the attributes are identical, the states are classified as the same abstract state. When the attributes differ, the states will be classified as different abstract states.

Before defining the abstract state for our validation application, the ING bankieren app, we looked at state of the art mobile scriptless testing and how they approach the abstract state problem. Four different approaches have been found:

- Use the Android Activity as abstract state [49].
- Same widgets on the screen (identical hierarchy tree of widgets) [50].
- Identical hierarchy tree of widgets and identical widget attributes [22, 51, 55].
- Identical screenshots [47, 54].

Using Android Activity as the abstract state does not work for dynamic Android apps as a lot of changes can occur within one Android activity. Identical screenshots would lead to no state abstraction as any change in the pixels of the screenshot results in a new abstract state. A tester-defined number of widget attributes to be identical seems to be the best approach for abstract state definition in the mobile domain.

For the ING application, we have opted to define everything which has identical widget class names to be the same abstract state (for other applications different settings can be selected). Essentially, if two states have the same class of widgets at the same location in the hierarchy tree, the states fall in the same abstract state group. E.g. text changing in the AUT does not change the abstract state or checkboxes being marked or unmarked does not affect the abstract state.

Note that how mTESTAR computes the comparison of the GUI states once it receives the attributes to compare, remains unchanged in this thesis (compared to TESTAR).

3.4.3 Abstract action

Using abstract states decreases the number of redundant actions being taken on the AUT, but it does not entirely prevent redundant actions. As we aim to minimize time and resource wasted, we introduce the concept of abstract action. To to best of our knowledge, no (scriptless) testing tool is working with a concept of abstract action.

In a state there are multiple actions possible on the GUI, some of them might have the same effect or be on the same sort of widget. Ideally, instead of taking all of these actions as separate unique actions, we group similar actions together, also called abstract action. When an abstract action has been executed, it will classify all of the actions in the abstract action group as executed. Figure 3.9 is an example of where the concept of abstract action can be valuable. All widgets marked by the red box result in a nearly identical state, only the name of the specific bank transaction changes. Having mTESTAR explore each of these actions and resulting states would result in a lot of wasted computation time.

For the best performance the definition of the abstract action is application specific. For the ING application, we have opted to apply abstract actions in two locations. The first location is in every GUI state which is related to the transactions. The second location is date pickers in the ING application. For transaction GUI states we have opted to let actions on widgets with the same Android class map to the same abstract action group. This is to prevent individual action interactions for every bank account or every transaction of a bank account. Additionally, we realized that mTESTAR can spend a large number of actions on date pickers as clicking on a day of the month is a specific action that can be repeated 31 times in the worst case. Abstract actions can also be applied more generally based on the widget attributes. However, for the ING application we have chosen not to define the abstract action more generally as we believe it would not improve the performance. Wrongly defining the abstract actions would also cause actions which lead to new states being skipped, limiting the exploration ability of mTESTAR.

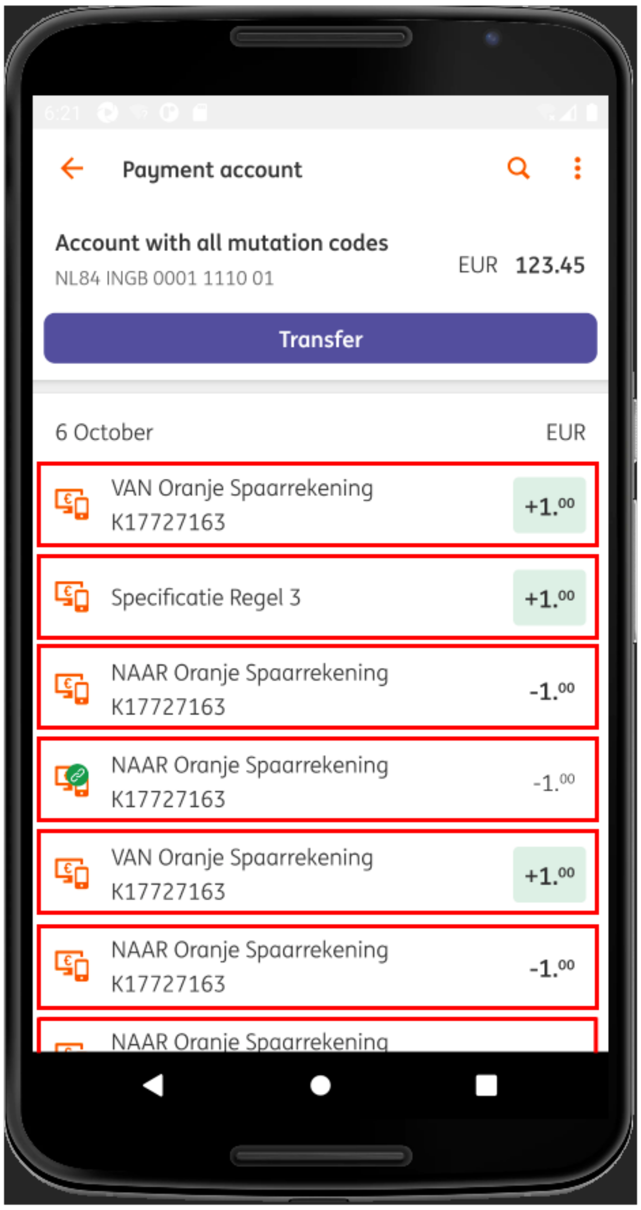


Figure 3.9: Screenshot of the GUI state of the Android ING Bankieren application where abstract action can be valuable.

3.5 Oracle

The different types of oracles, and the oracles in existence for automated testing have been introduced in Section 2.3. Using these concepts we have extended the test oracle within mTESTAR in an attempt to create oracles that are better at detecting incorrect behavior. In order to improve the fault detection of the mTESTAR mobile scriptless testing tool, it is important to not only design the individual oracles detecting different problems but also to create a framework in which the oracle results can be combined. Additionally, testers should be able to define oracles themselves for verifying application-specific requirements. To achieve both we introduce an oracle framework for mTESTAR.

We define the requirements for individual oracles in this framework as follows:

- Oracles shall be able to be composed to allow for combining the results of different oracles.
- An oracle shall be atomic; the oracle at most takes care of one type of fault.
- Oracles shall be deterministic; the oracle will always produce the same output for the same input.
- Oracles shall be able compute their output within 3.0 seconds of receiving the GUI state.
- The oracles shall be generic; aim to require as little maintenance as possible (when the AUT changes).
- The oracle shall output a number between 0-1 specifying the likelyhood of a potential fault, where 0 indicates no faults were found and 1 indicates a (fatal) crash was encountered.

For the composition of the oracles, we require that the tester can specify how the outputs (value between 0-1) of the oracles are combined into a final output value which is translated to a verdict. A verdict is one of the following three statements: (OK, WARNING, ERROR) together with the string description of the oracle findings. In Figure 3.10 it can be observed how the oracle framework with the composition of oracles would be positioned within the mTESTAR execution flow.

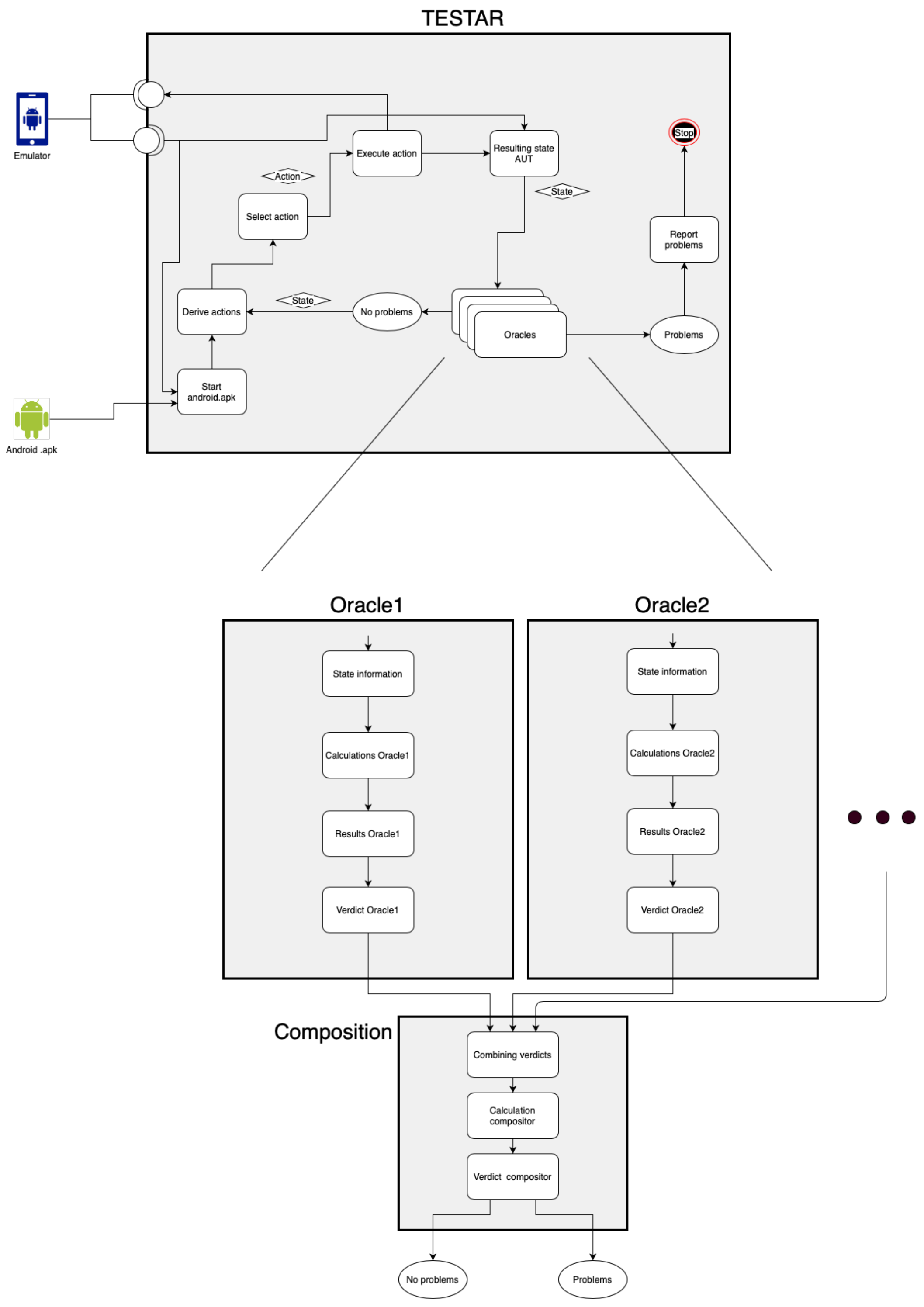


Figure 3.10: The mTESTAR execution-flow with a visualization of how the oracle and oracle composition takes place within this execution flow.

3.5.1 Devised oracles

The individual oracles we devised and determined valuable for detecting incorrect behavior during scriptless testing are as follows:

- Crash oracle

- Log oracle
- Suspicious text oracle
- Language oracle
- Deadlock oracle
- Livelock oracle
- Profiler oracle
- Model difference oracle

The crash oracle is an implicit oracle and monitors the AUT and the mobile device on which the application is running. If a fatal error occurs (stopping the AUT or crashing the mobile device) the crash oracle logs the information and returns that a fault has been found. This is a simple oracle that can be found in all scriptless testing tools which have a fault detection component.

The log oracle is an implicit oracle and monitors logs send by the operating system of the mobile device. The log oracle records information about any warning or errors with as subject the AUT. Based on the severity of the warning or error it reports a different verdict score. The log oracle can be found in some mobile scriptless testing tools (e.g. Stoat and Sapienz).

The suspicious text oracle is taken from TESTAR for desktop and web applications. In this specified oracle regular expressions are specified, the oracle will check for every GUI state whether these regular expressions are mentioned anywhere in the widgets. If there is a match the oracle logs the widget information in which the suspicious text can be found and returns a verdict with this information.

The language oracle [64] is an implicit oracle focused on the accessibility of the AUT. To the best of our knowledge, this type of oracle has not been implemented in the scriptless testing for mobile application. The language oracle requires the tester to specify the language setting of the AUT. The oracle proceeds to scan all text present in the GUI on spelling errors encountered by comparing the text to a dictionary. Additionally, the language oracle verifies if there are no language inconsistencies in the application like missing translations. If any spelling or translation issues are encountered the oracle records the information in its verdict.

mTESTAR makes use of a state model to record what GUI states and actions it already has encountered. If mTESTAR gets stuck in the same, or a group of similar states it could indicate that that the user can get stuck in a certain part of the GUI without having the option to return or progress to other states. This is undesirable behavior for most GUIs. To detect this fault the implicit deadlock and livelock [65] oracle are devised. The deadlock oracle verifies the application is not stuck in one specific GUI state. The livelock oracle verifies the application is not stuck in a small set of GUI states. To the best of our knowledge, there exists no oracle for scriptless testing tools checking for deadlock and livelock.

The profiler oracle monitors the CPU, memory, network, energy consumption of the mobile device running the AUT. This oracle checks if the device does not exceed the maximum values specified by the tester for any of the aforementioned properties. Additionally, the oracle monitors if a peak in any of the monitored properties occurs when the GUI changes. E.g. the network should not be maximally loaded at page switch, or the CPU heavily used when the GUI page is scrolled.

Lastly, we have the model difference oracle. This is a human oracle. All previously introduced oracles are online oracles, the model difference oracle is an offline oracle. The goal of the model difference oracle is to support the tester in comparing different AUT versions. The tester specifies which versions of the AUT should be compared, and the model difference oracle will report on the differences in the abstract and concrete state models mined. The tester can use this report to determine if any unexpected changes occurred. State model difference can be computed through the PLTSDiff algorithm devised by Bogdanov *et al.* [66]. Additionally, the value of this oracle has been confirmed by software testers of the ING bank. They indicated this oracle would help them out when new features are introduced into the application.

Note that the profiler oracle and the model difference oracle have not been completed for mTESTAR. The language oracle has partly been implemented for mTESTAR. However, all three oracles would provide further issue detection capabilities to mTESTAR.

3.5.2 Oracle composition

To allow testers to combine the different oracles (self-designed oracles or predefined mTESTAR oracles) a domain-specific language (DSL) has been designed in which the combining strategy desired by the tester can be specified. The idea behind the DSL is to construct a hierarchical tree of the oracles, which need to be constructed from the code the tester submits. In the tree, leaves are the oracles and intermediate nodes are combined verdicts, see Figure 3.11 for an example. This tree corresponds to the DSL code in Listing 3.1:

Listing 3.1: DSL code example

```
[SpellcheckOracle][CrashOracle,LogOracle]
[SuspiciousTextOracle][DeadlockOracle,LivelockOracle]
```

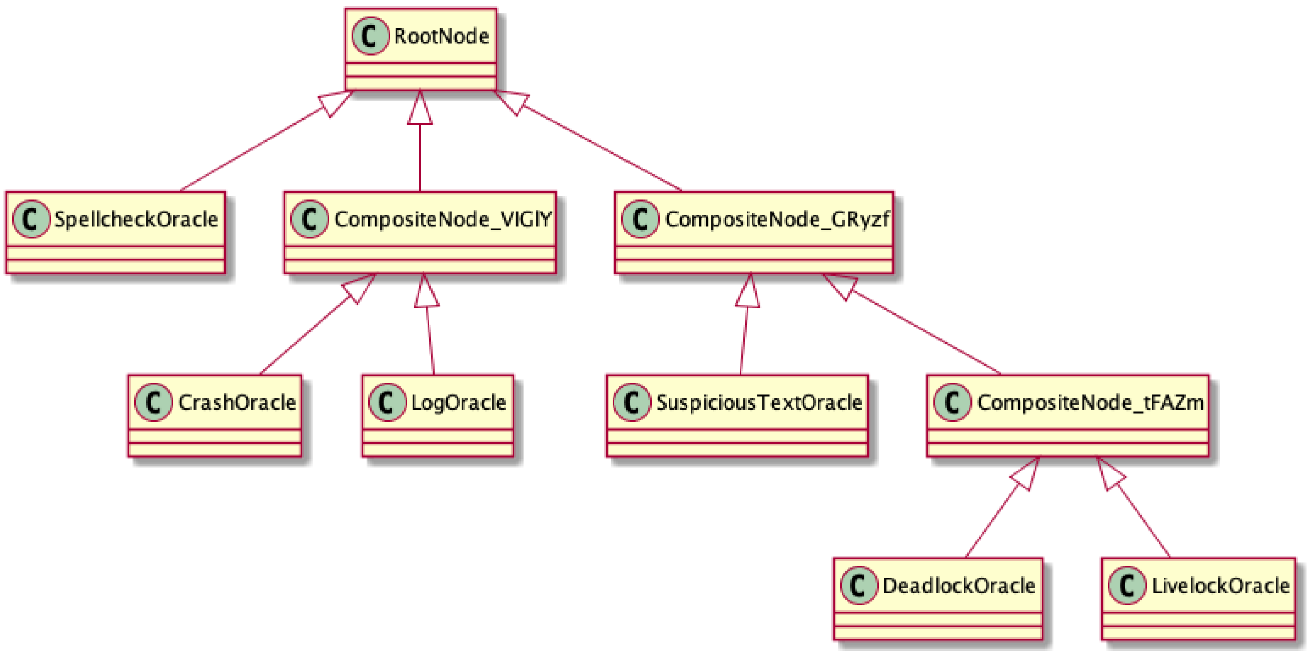


Figure 3.11: The visual tree representation of the oracle composition tree created by mTESTAR. The diagram is generated from DSL code through plantUML.

In the DSL the tester specifies oracles that it directly wants to combine between squared brackets, separated by a command (*[oracle1,oracle2]*). Oracles that need to be combined with this composite result are specified on the same line within squared brackets (*[oracle1,oracle2][oracle3]* or *[oracle1,oracle2][oracle3][oracle4]*). If there is more than one composite result that has to be combined, one of the compositions must be specified on a separate line, see code block 3.1. There is no limit to the number of lines that can be used in the DSL.

The composition function used in combining oracle results can be specified by the tester. mTESTAR supports two approaches; max combiner and threshold combiner. In the max combiner, the oracle result with the value closest to one is the result passed to the combined oracle verdict. When there is more than one oracle result with the same max value all of these results are passed. In the threshold combiner the tester defines a value between 0-1, all oracle results greater or equal to this value are stored and passed up in the tree as oracle result.

Visualizing the oracle composition tree allows the tester to verify if the oracle tree specified in the DSL is the oracle composition intended. The visualization of the tree happens through the PlantUML open source project[5].

3.6 Conclusion

This chapter presents the mTESTAR tool and the oracles designed. mTESTAR allows testers to use scriptless GUI testing for mobile applications. Additionally, an oracle framework is designed within mTESTAR. The framework allows testers to specify the oracle compositions and creating new oracles. mTESTAR implements the design aspects extracted from the literature research.

Overall, mTESTAR answers the question of how TESTAR can be extended for Android and iOS applications. Additionally, mTESTAR also answers how testing oracles can be used for scriptless testing.

[5]https://plantuml.com

Chapter 4

Industrial validation

In this chapter, we will present the performance of mTESTAR on the ING Bankieren application in terms of the code coverage metric. ING is a well-established bank[1], which has a significant IT component[2] due to the current-day importance of software in (E-)banking [67]. ING Bankieren is the official application to access the bank's information for Dutch ING clients. Additionally, the ING Bankieren application is downloaded over 5 million times[3]. Therefore, we select the ING Bankieren application for validating the performance of mTESTAR, given that it is an industrial and in-use application.

Presenting the performance of mTESTAR in isolation provides no comprehensive idea about the effectiveness of the tool. Instead the performance of mTESTAR should be compared to a baseline. Therefore, we have also measured the code coverage achieved by the Espresso GUI tests designed by the Android development team of the ING bank. Espresso tests are scripted tests designed for testing the GUI of an Android application. Comparing the Espresso result and the mTESTAR result is of great value. From this comparison, we can determine the value of mTESTAR for industrial use and evaluate whether mTESTAR's scriptless testing can outperform or equal the performance of the scripted testing.

Additionally, to determine how mTESTAR performs versus other state-of-the-art scriptless testing approaches, we evaluate Stoat and DroidBot (see Chapter 2) on the same ING application. The comparison allows us to evaluate the design choices made.

4.1 Code coverage

To evaluate the performance of the designed and implemented mTESTAR GUI scriptless testing tool, we use the performance metric of code coverage. Code coverage is a commonly used testing performance metric [68] and many of the scriptless testing tools mentioned in Section 2 use it as a performance metric. The concept behind code coverage is that the code which is covered does not have issues. If the covered code contained faults, they would have shown during the tests covering the code. Therefore, more code covered implies more potential faults found.

To determine the performance of the mTESTAR implementation, we will measure its performance on the Android ING Bankieren application. We need an additional tool for measuring the code coverage. As Android is Java-based, we can use Java code coverage tools to measure the code coverage. The tool we have opted to use is Java Code Coverage (JaCoCo). JaCoCo is an open-source project which has no restrictions in terms of allowed use. As Android is developed through Java and the Android SDK,

[1]https://www.ing.com/About-us/Profile/History.htm
[2]https://www.ing.com/Newsroom/News/Nothing-beats-engineering-talent-INGs-agile-transformation.htm
[3]`https://play.google.com/store/apps/details?id=com.ing.mobile`

JaCoCo can be leveraged to obtain code coverage on the ING Bankieren application. Note that the EMMA [53] and ELLA [52] code coverage tools have been investigated and applied to the Android ING Bankieren app given their prominence in the literature concerning code coverage. However, both tools are unmaintained and did not function as intended on the Android ING Bankieren application, therefore we will not incorporate them in our evaluation.

JaCoCo works with the concept of instrumentation on the byte code level. JaCoCo creates instrumented versions of the original class definitions to keep track of what has been executed. It achieves this by attaching itself to the JVM and instrumenting the classes during runtime at class loading. Using the instrumentation, JaCoCo can determine which classes are called and what instructions are executed. To translate this information into coverage reports, JaCoCo needs access to the source code. Byte level coverage does not directly translate to the same source code coverage [69]. Using the source code it obtains details on the classes and which instructions are on each line. This results in the coverage on package, method, line, and instruction level. JaCoCo working at build time requires that the source code must be available to set up JaCoCo. This limits JaCoCo's use to applications of which the source code is accessible to the tester.

Although Android is Java based, Android and Java are not identical. Additionally JaCoCo is designed to work for predefined test cases. In scriptless testing, test sequences are not predefined but instead are dynamically constructed. To help with setting up JaCoCo for Android applications and support code coverage for dynamically constructed test sequences, we use the COSMO tool [69]. COSMO installs the JaCoCo tool on the source code of the application for which the code coverage must be measured. Broadcasts are sent to the mobile device to obtain the code coverage dynamically. These broadcasts result in coverage files that are passed back to JaCoCo to generate reports.

We evaluate the code coverage over the whole ING Bankieren application and investigate the code coverage at the package level. With the use of JaCoCo, we obtain the code coverage achieved by the ING Espresso tests. With both JaCoCo and COSMO, we obtain the code coverage of mTESTAR, Stoat, and DroidBot on the Android ING Bankieren application.

4.2 Experiment setup

When evaluating the performance of mTESTAR, we test for different settings: exploration algorithm, number of test sequences, and the number of actions per test sequence. For the exploration algorithm, we test the random selection algorithm, the unvisited actions first algorithm, and the Q-learning algorithm as these are supported in mTESTAR. Before evaluation, the optimal performing settings for the number of sequences and sequence length are unknown, so multiple combinations have been tried. The total number of actions in any mTESTAR run is the same; 3000 actions, to ensure that the different settings can be compared. We tested the combinations of; 6 sequences - 500 actions, 10 sequences - 300 actions, 30 sequences - 100 actions, 50 sequences - 60 actions, and 100 sequences - 30 actions.

All experiments have been carried out on an Android emulator running on the same device to ensure the hardware does not affect the test results. See Appendix A for the detailed system information.

The ING Android developers create the Espresso tests. As Espresso tests are scripted no settings can be modified. Stoat and DroidBot do not work with test sequences. Both tools test until a limit for the total number of actions is reached. To achieve a fair comparison between mTESTAR, Stoat, and DroidBot, the number of actions for Stoat and DroidBot is also set to 3000.

4.3 Results

To measure and compare the performance for Espresso, mTESTAR, Stoat, and DroidBot, we present the code coverage over the whole Android ING Bankieren application. For mTESTAR, we list the performance for the combination of all settings introduced in Section 4.2. For an overview of the performance see Table 4.1. The best performing setting for mTESTAR is 10 sequences, 300 actions, and the Q-learning algorithm with a code coverage between 40.8-41.9%. This code coverage is slightly worse than the performance achieved by Espresso. However, mTESTAR achieves significantly better performance compared to Stoat and DroidBot.

Table 4.1: The table with the coverage results for the Android ING Bankieren application.

Tool	Algorithm	Sequences	Actions	Instructions Covered	Lines Covered	Methods Covered
mTESTAR	Unvisited action	10	300	34.9%	34.5%	35.8%
mTESTAR	Unvisited action	30	100	37.2%	36.8%	38.0%
mTESTAR	Unvisited action	100	30	37.0%	37.0%	38.5%
mTESTAR	Unvisited action	60	50	34.2%	34.0%	35.4%
mTESTAR	Unvisited action	6	500	35.2%	35.1%	36.7%
mTESTAR	Random action	10	300	40.2%	39.9%	40.2%
mTESTAR	Random action	6	500	36.7%	36.7%	37.7%
mTESTAR	Random action	30	100	36.2%	35.6%	37.0%
mTESTAR	Random action	100	30	35.6%	35.5%	37.1%
mTESTAR	Random action	60	50	38.1%	38.3%	39.3%
mTESTAR	Q-learning	10	300	41.0%	40.7%	40.8%
mTESTAR	Q-learning	30	100	37.7%	37.9%	38.8%
mTESTAR	Q-learning	100	30	39.1%	39.0%	40.0%
mTESTAR	Q-learning	60	50	38.4%	38.5%	39.5%
mTESTAR	Q-learning	6	500	40.4%	40.3%	40.8%
ING scripted tests	-	-	-	43.9%	43.4%	45.9%
Droidbot	Unvisited actions first	-	-	34.9%	34.1%	35.7%
Droidbot	Random	-	-	34.3%	33.7%	35.3%
Stoat	-	-	-	23.7%	22.6%	25.7%

As the exploration algorithms used by mTESTAR are non-deterministic, the coverage results for runs with the same settings can differ. Due to the time restrictions, it is infeasible to run each mTESTAR setting combination until the central limit theorem [70] can be applied. Table 4.2 shows the recorded runtime for each setting of mTESTAR. Note that runtimes recorded are subject to variance as the runtime is not averaged over multiple mTESTAR executions.

Table 4.2: The table with the time performance for the different settings of mTESTAR.

Algorithm	Number of sequences	Number of actions	Time taken
Unvisted action	10	300	274min
Unvisted action	30	100	301min
Unvisted action	100	30	340min
Unvisted action	60	50	303min
Unvisted action	6	500	248min
Random action	10	300	257min
Random action	30	100	277min
Random action	100	30	350min
Random action	60	50	318min
Random action	6	500	270min
Q-learning	10	300	228min
Q-learning	30	100	263min
Q-learning	100	30	348min
Q-learning	60	50	297min
Q-learning	6	500	237min

Assuming the central limit theorem can be applied after obtaining 30 results [71], with a total of 15 different settings and an average runtime of 287minutes, the total computation time required would be 2,152.5 hours or approximately 90 days of continuous running. This runtime is infeasible for the scope of this thesis. Therefore, it has been opted to only run the best performing mTESTAR settings ten additional times.

The extra mTESTAR runs will allow for observing the variation in the result caused by the non-determinism of the exploration algorithm. Additionally it can be used to validate that the result is not caused by randomness. The results of the additional runs for Q-learning: 10 sequences - 300 actions, are presented in Table 4.3.

Table 4.3: The table with the coverage results on the Android ING Bankieren application with the Q-learning algorithm, 10 sequences, and 300 actions per sequence.

Run number	Instructions Covered	Lines Covered	Methods Covered
Original run	41.0%	40.7%	40.8%
1	39.8%	39.7%	39.9%
2	39.5%	39.5%	40.0%
3	40.1%	40.3%	40.6%
4	40.9%	40.7%	40.9%
5	40.0%	40.3%	40.7%
6	40.8%	41.0%	41.1%
7	41.0%	41.3%	41.6%
8	40.5%	40.4%	40.8%
9	39.8%	39.5%	39.9%
10	40.3%	40.4%	40.8%
AVERAGE	40.4%	40.3%	40.7%
VARIANCE	0.29	0.34	0.28

From the results in Table 4.3, we observe that the average over ten runs is slightly lower than the original performance recorded. Additionally, the variance observed is low, indicating the randomness in the Q-learning algorithm does not affect the overall coverage results. Together it allows us to claim with more confidence that mTESTAR achieves performance very close to the Espresso tests. Note that non-determinism in the Random exploration and unvisited action first algorithms is greater. Therefore, we expect that the results differ more between runs for these algorithms. Additional experiments will have to be carried out to confirm this expectation.

It could be the case the mTESTAR runs achieve higher coverage given more time. To validate the number of sequence settings tested, we present the coverage increase per sequence. If the code coverage increase flattens, it indicates more test sequences would not significantly improve the code coverage. See the code coverage over sequences for the different exploration algorithms in Figure 4.1, Figure 4.2, and Figure 4.3.

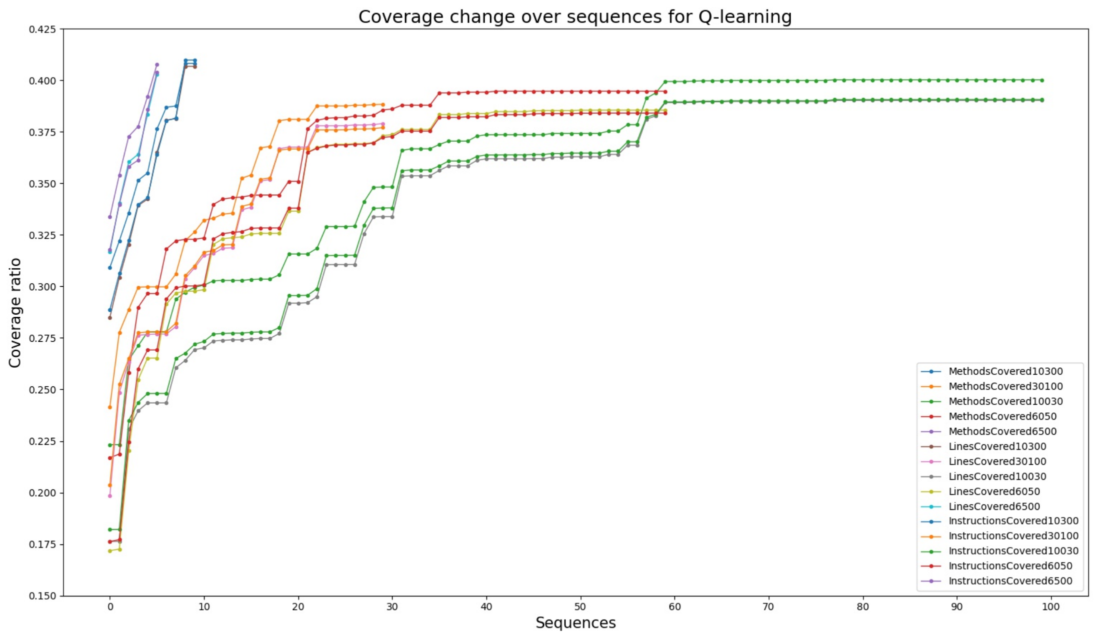


Figure 4.1: Figure showing the coverage over sequences for Q-learning and the different mTESTAR settings.

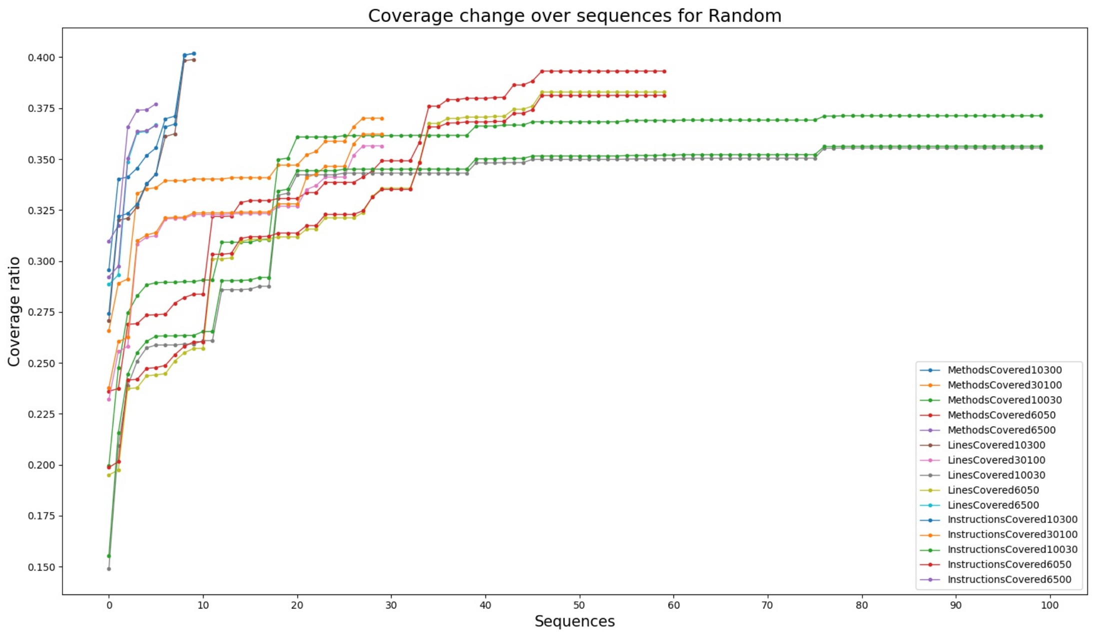


Figure 4.2: Figure showing the coverage over sequences for Random and the different mTESTAR settings.

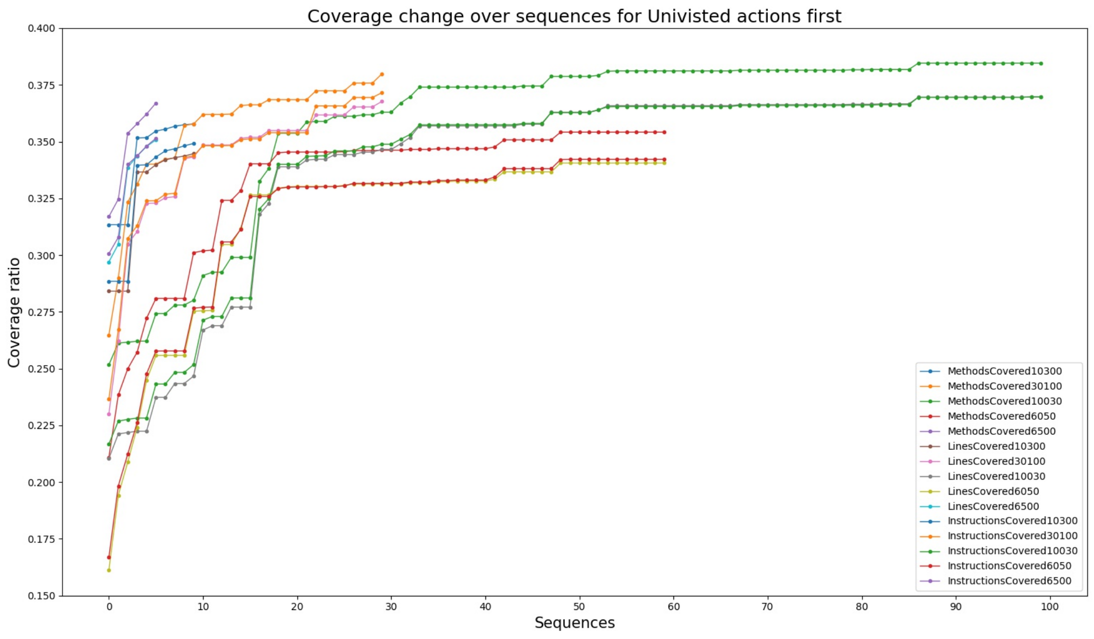


Figure 4.3: Figure showing the coverage over sequences for Unvisited actions first and the different mTESTAR settings.

From Figures 4.1, 4.2, and 4.3 we observe that for the settings; 30 sequences - 100 actions, 50 sequences 60 actions, and 100 sequences - 30 actions, the curve for the code coverage has flattened. This indicates increasing the number of sequences for these settings would not significantly increase code coverage. For 10 sequences - 300 actions and 6 sequences - 500 actions, it can be argued that the curve has not flattened, and an increase in sequences could improve the performance. To validate if the performance can be increased the best performing algorithm for both these settings, Q-learning, is selected and the number of sequences is increased. The performance of these mTESTAR runs is presented in Table 4.4. As can be observed, the results are similar to the non-extra sequences runs of mTESTAR, indicating additional sequences do not improve the code coverage.

The code coverage increase halting when running more sequences (increasing the overall actions executed) is unexpected. Expected is that given enough time, exploration will reach more (border) cases and increase the code coverage. However, there are possible arguments for a code coverage lower than 100%. Firstly, 100% coverage is infeasible is because the ING Bankieren application has a significant amount of code that is inaccessible. The code is either no longer called or legacy code. We could not filter out all these code components as they are sub-components of classes, and filtering on method level proved impossible. However, the inaccessible code is not 50% of the total code, indicating there must be other factors affecting the code coverage of mTESTAR not growing past 45%.

A potential second factor limiting the code coverage obtainable is that the ING Bankieren application ran in a test environment as connecting to the live environment was not permitted for the testing of mTESTAR. ING Bankieren developers do indicate this should not significantly affect the code possible to be covered. Further tests must be carried out for finding additional factors limiting the code coverage obtainable by mTESTAR.

Table 4.4: The table with the coverage results of additional mTESTAR runs to verify the settings chosen for mTESTAR.

mTESTAR	Instructions Covered	Lines Covered	Methods Covered
15 Sequences, 300 Actions Q-learning	40.7%	40.2%	40.7%
10 Sequences, 500 Actions Q-learning	40.1%	40.0%	40.5%

4.3.1 Package level results

The results presented previously are the coverage results over the whole ING Bankieren application. As the performance of the Espresso tests and mTESTAR are similar, it is interesting to examine the performance of mTESTAR versus Espresso in more detail. To obtain a more detailed comparison, we investigate the code coverage at the package level. For each package we record the code coverage of mTESTAR and subtract the code coverage of the Espresso test. The expectation is that most packages will have similar coverage for both mTESTAR and Espresso, thus the difference will be approximately zero. The expectation is confirmed by the violin plots of Figure 4.4.

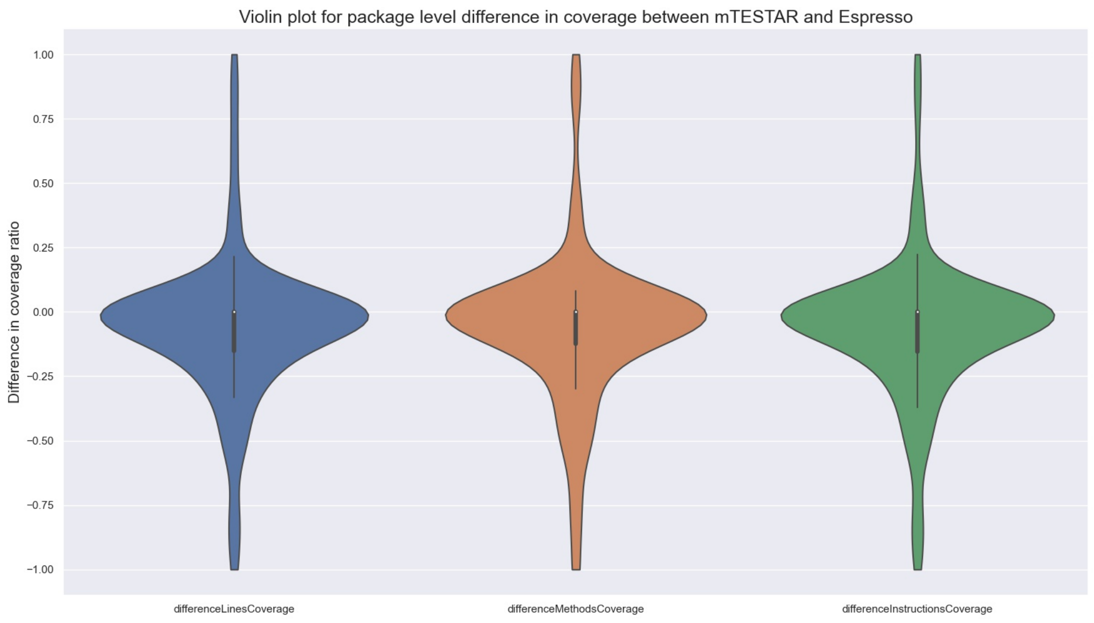


Figure 4.4: Violin plot for package level difference in coverage between mTESTAR and Espresso

The density of all three violin plots is near the 0.0 mark, indicating both mTESTAR and Espresso achieved similar coverage for most packages in ING Bankieren. This confirms the result that mTESTAR works as well as scripted testing (Note that the Espresso tests are the actual tests used by ING before release). However, there are some packages near 1.0 and −1.0, indicating mTESTAR or Espresso achieved a much better coverage compared to the other. The top 20% of the packages concerning code coverage difference is displayed in Figure 4.5. Note that again for each package the code coverage of Espresso is subtracted from the code coverage of mTESTAR. Additionally, the names of the packages are anonymized as the names contain confidential ING information.

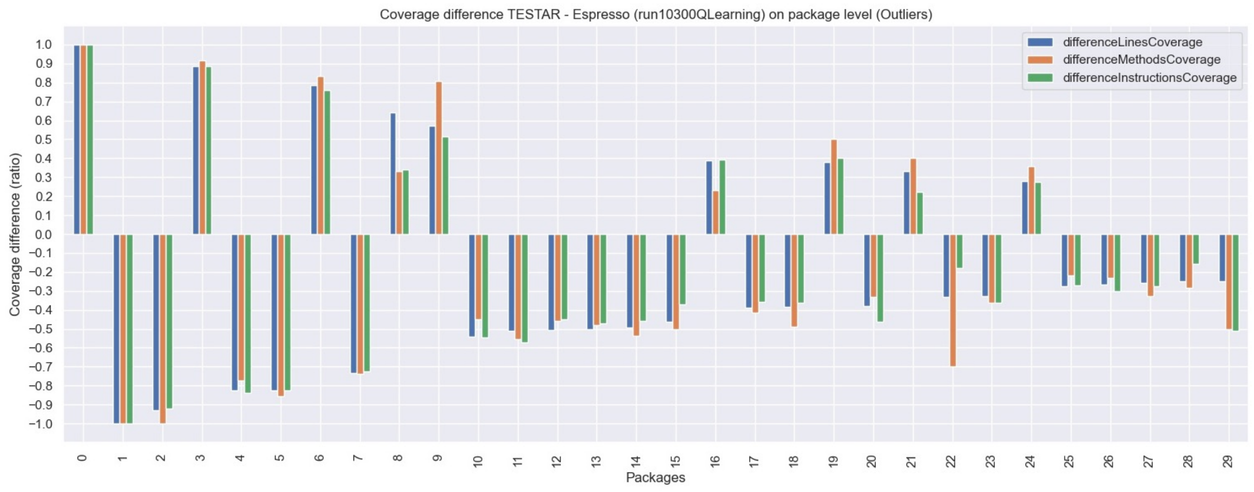


Figure 4.5: Figure showing the 20% of the packages with respect to the difference in coverage between mTESTAR and Espresso.

From Figure 4.5, it can be observed that the absolute value of the outliers decreases reasonably quickly. Examining the top 10% shows that the difference in coverage has already decreased until approximately 0.5. Evaluating the characteristics of the packages where mTESTAR outperforms Espresso (or vice versa) would allow for detecting possible relations between these packages. Therefore, we will perform an analysis of the outlier classes.

The packages containing operations and settings modifying the information available to ING users pre-login are in the outliers, mTESTAR achieved better code coverage for these packages. These packages contain the code relevant for customers to modify what is visible pre-login but also the code relevant to displaying the actual information to the customer. mTESTAR is capable of covering this code as it able to log-out within the application and continue exploring the AUT.

The second characteristic of packages where mTESTAR outperforms Espresso is related to exporting. From the ING application information related to bank accounts can be exported. The packages containing the operations for exporting information are in the outliers concerning code coverage differences. Specifically, mTESTAR covers the code related to generating PDF's with different types of information. Depending on the type of information to be exported different PDF generating code is called. mTESTAR seems capable of performing the export function from multiple contexts.

As the performance on the complete ING Bankieren application is better for the Espresso tests, there are numerous packages where Espresso outperforms mTESTAR. Espresso achieves greater code coverage for all packages related to security. A number of the security packages are in the outliers of Figure 4.5. Additionally, the packages for login obtain higher code coverage through the Espresso tests. mTESTAR has a predefined login sequence setup to get into the ING Bankieren application reliably. This could be the cause of the lower code coverage achieved for the login packages. However, it does not explain the worse performance of mTESTAR for the security packages.

Due to the difference in coverage between packages, it is interesting to determine what the code coverage is if Espresso and mTESTAR are used together. Using JaCoCo, we obtain the following code coverage for Espresso and mTESTAR

- **Instructions Covered:** 52.3%
- **Lines Covered:** 52.1%
- **Methods Covered:** 52.3%

This is a very significant increase in coverage, even compared to Espresso. Overall, Espresso and mTESTAR cover different aspects of the ING Bankieren application, where the best coverage is achieved by using them together.

4.4 Discussion

From the results we observe that mTESTAR outperforms both Stoat and DroidBot in terms of code coverage for the ING Bankieren application. Therefore, we claim mTESTAR adequately implements the design aspects extracted from the related scriptless testing tools, see Section 2.6.1. Additionally, as it outperforms both Stoat and DroidBot, we can consider mTESTAR to have state of the art scriptless testing performance for the industrial ING application.

It can be argued domain-specific knowledge has been added to mTESTAR, causing it to perform better than DroidBot and Stoat (mTESTAR has the abstract state and abstract action setup for ING Bankieren). However, both DroidBot and Stoat are supplied with domain-specific information as well. For Stoat and DroidBot, a login sequence and the filtering of widgets are added as domain-specific knowledge. To ensure Droidbot does not escape the AUT, it has additional predefined actions specified. As both Stoat and DroidBot do not support adding any additional domain-specific information (E.g. no abstract state or abstract action), the comparison between the tools can be justified.

It must be noted that Stoat and DroidBot only use a subset of the available scriptless testing approaches (e.g. we did not compare performance versus a tool like SIG-Droid using symbolic execution to explore the GUI). To be able to claim the design choices and settings used in mTESTAR achieved the best performance of all possible tools, comparison versus more state of the art tools is required.

The central limit theorem could not be applied to the mTESTAR performance obtained. Running all tests enough times to get normally distributed results would take too much time with the available resources. However, it is shown that the randomness of the Q-learning algorithm employed is limited, and there is a low variance in the best performing mTESTAR setting. Together with the observation mTESTAR is within 2% of the Espresso performance, we can conclude mTESTAR's performance is comparable to the scripted Espresso tests on the ING Bankieren application.

Although mTESTAR and Espresso achieve similar code coverage results, it is complicated to fairly compare scripted and scriptless testing. For Espresso tests, it is still relatively simple to leverage the domain knowledge of the tester in the scripts by having the expected results coded in. This allows for testing at a functional level. For mTESTAR, the ability to test whether the AUT meets the requirements must happen through the oracles. Creating oracles with all the domain knowledge required to test at the same functional level as scripted testing would massively increase maintenance costs, essentially removing the advantage of scriptless testing having less resource (time and maintenance effort) consumption. Instead, mTESTAR is well suited to test for implicit requirement breaches. Implicit oracles require little maintenance and mTESTAR itself as well. Therefore, the test created in Espresso searching for implicit breaches of the requirements is best transferred to mTESTAR. Overall, we believe mTESTAR has value in the process of testing industrial applications but should be used together with scripted (Espresso) tests.

From the results it can be observed mTESTAR takes a significant time to execute the test sequences. Especially considering the Espresso tests run for approximately an hour. This difference in runtime can be an important factor affecting where in the development cycle mTESTAR can be used. For nightly builds or release builds the runtime for mTESTAR should not be an issue as they can be run in parallel with any other tests. However, running mTESTAR every time a developer wants to integrate its changes may significantly slow down the development cycle. Do note that although Espresso runs quicker, it does require a significant amount of time to design and create the tests, opposed to mTESTAR which only has the runtime.

In conclusion, we believe mTESTAR has value in the process of testing the ING Bankieren application. However, it cannot just replace the scripted testing. mTESTAR tests with a different scope (not suited for functional testing) and the coverage of mTESTAR is different than the coverage of Espresso. Using mTESTAR and Espresso together is the approach in which mTESTAR adds value to the testing process.

Chapter 5

Threats to validity and Future work

This chapter consists of three sections. Section 5.1 highlights the risks to the validity of the research and how we have attempted to mitigate them. Section 5.2 presents possible future work directions. Section 5.3 contains personal recommendations for ING.

5.1 Threats to validity

Runeson *et al.* [72,73] present a structure for evaluating the threats to the validity of software engineering research. We apply this framework to our work to evaluate the threats.
Runeson *et al.* introduce four types of threats that can affect the validity of software engineering research;

- **Construct validity:** The construct validity evaluates to what degree the measures studied accurately represent what is investigated according to the research questions. To evaluate the performance of the testing methods used, we use the metric of code coverage. The problem with the code coverage metric is that it does not directly relate to the ability of the testing tool to find faults within the software. It can be the case all bugs are concentrated in one method of the code. Testing everything but this method would result in very high code coverage but not discover any faults. Having tests only cover the method with all bugs would have low code coverage but find all problems. The low code coverage test thus has better performance for fault detection. However, as it is unknown beforehand if and how many faults are present in the software, code coverage is generally accepted as the best solution for measuring the performance of tests.

- **Internal validity:** The internal validity is concerned with to what extent the observed results represent the truth in the studied population. To mitigate this threat, we evaluate which components of the code-base should be covered by the tests together with the ING Bankieren developers. This ensures the code coverage is only measured for the packages the tool should be testing. Additionally, JaCoCo is verified by manually comparing the result for identical test sequences and checking the code coverage is identical.

- **External validity:** The external validity is concerned with to what extent the results and findings can be generalized. Additionally, it is concerned to what extent the findings are of value to people outside the investigated case. We validated mTESTAR on one industrial application; the ING Bankieren app. Although this is an industrial application actively used by millions of people, it is only a single application. Additional testing needs to take place to prove mTESTAR works for other industrial applications. However, mTESTAR has been designed such that it is a tool generally usable. The domain-specific information added to support testing the ING Bankieren application is easily adjustable.

- **Reliability:** The reliability is concerned with the degree to which the researchers affect the results. Ideally, if the study is repeated, the results should be the same. The reliability is a significant threat to the validity of the thesis. For most of the settings of mTESTAR, the tool is only executed one or two times. As the algorithms used for exploration are non-deterministic, there is a chance the results will vary when running mTESTAR again. Due to the long-running times, it was infeasible to run all settings multiple times such that the central limit theorem could be applied. Therefore, the reliability threat could not be completely mitigated. However, the best performing setting of mTESTAR has been executed an additional ten times showing low variance. This indicates the best performance recorded can consistently be achieved, mitigating the reliability threat somewhat.

5.2 Future work

There are four future work directions for the scriptless GUI testing tool mTESTAR;

- **Oracles:** Future work can focus on improving the oracles for scriptless GUI testing in the mobile domain. We have designed and implemented some oracles in this thesis, most of which are implicit oracles. A deeper look at specified oracles could potentially lead to more oracles for scriptless testing. Additionally, with the growing possibilities of machine learning, this avenue can also be explored to develop new oracles. An example would be collecting a dataset of Android applications showing unexpected behavior. A supervised machine learning algorithm could then learn from this dataset and be taught to monitor an application and report abnormal behaviors.

- **iOS:** mTESTAR for iOS can be improved. Currently, Appium is used as accessibility API. However, Appium has a bottleneck; it is slow in returning the GUI state for iOS. Specifically, it takes Appium 2000ms on average to return the GUI state. As mTESTAR often requests the GUI state, the overall testing is significantly slowed down. Having a very long testing process every time the GUI tests need to run is highly undesirable. Therefore, improving the iOS testing speed would be a valuable improvement to mTESTAR.

- **Data generation:** The *type* action has room for improvement. The *type* action currently either enters a randomly generated input or enters an input from a predefined list. As almost every application has input fields that can be interacted with (possibly affecting the following up actions that can be executed on the GUI), it would be valuable to have the possibility to generate input related to the domain of the application. An avenue that can be explored is generating synthetic data from the input of real users[1]. Overall, allowing the input data to resemble real input improves the ability to explore an application.

- **Validation of mTESTAR:** It would be valuable to extend the validation of the performance of mTESTAR. The validation can be improved in two ways; different metrics or more experiments. The current validation is focused on the metric of code coverage. Although code coverage is a generally accepted metric to measure the performance of tests, it does not report if and what the faults found within the AUT are. Validating mTESTAR on an application with known bugs would be valuable to determine how reliably mTESTAR can find them. As mentioned, the mTESTAR runs

[1]https://news.mit.edu/2020/real-promise-synthetic-data-1016

take a long time and we were unable to obtain enough results to apply the central limit theorem in the available time. Completing the runs for the presented mTESTAR settings would strengthen the claims made.

5.3 Personal recommendations for ING

My graduation project at ING was kicked off by the question of how ING could improve its GUI testing for Android and iOS applications. Specifically through testing methods that do not involve scripted (user-story based) testing. Through the experience obtained during this thesis, I would like to mention three suggestions.

The first recommendation is to continue working with teams developing the ING application and allow them to generate new idea's for oracles. During the thesis, I came in contact with multiple teams and engineers who had valuable inputs. The engineers allowed us to understand what they deem important to test, where problems occur, and what helps them do their testing. As the developers have first-hand experience with the bugs that can arise in the applications, this is valuable information. The next step in the process would be facilitating the developers in creating their own oracles. Additionally, the oracles are an important factor for the fault detection capabilities of mTESTAR. Thus having them well defined is crucial.

The second recommendation is to open source the non-ING-specific components of mTESTAR. Allowing the testing community to improve the core components of mTESTAR leads to additional resources and ideas being contributed to mTESTAR. Original ideas can benefit the performance of mTESTAR and thus benefit ING in their testing process as well. By only releasing the non-ING-specific information as open-source, no confidential information is leaked.

The last recommendation is to closely evaluate the iOS performance and determine if there is a need for a different accessibility API. Although mTESTAR functions for iOS, it is slow. When there are time restrictions for the tests, slowness can limit the usability of mTESTAR. If it is possible to speed up mTESTAR for iOS, it could prove more valuable for the ING iOS development teams.

Chapter 6

Conclusion

Software development is a giant industry and continuously evolving. As organizations and people increasingly rely on software, it is important the software developed meets its requirements. Software testing is an important aspect of the software development cycle ensuring the quality of applications. GUI testing is crucial within software testing as it allows for end-to-end testing of the AUT and tests the user-facing part of the AUT. Traditional approaches to GUI testing (manual and scripted testing) are expensive and time intensive. To address this resource issue, the concept of scriptless testing has been conceived. In scriptless testing, algorithms are used to automatically generate test sequences in an attempt to lower maintenance requirements. As the mobile platform is becoming increasingly important, it provides an opportunity to apply scriptless testing for mobile applications.

This research aims to design and implement a scriptless GUI testing tool for mobile applications and validate it on an industrial application. To guide the work we formulated research questions.

The first research question we propose is; "What is the state of the art in the scriptless GUI testing for mobile applications?". We answered this question through a literature study in Chapter 2. The literature study covers the need for automated GUI testing, the GUI exploration algorithms used in scriptless testing tools, introduces oracles for automated GUI testing, and evaluates state of the art tools for mobile scriptless testing. We found a number of state of the art tools that can be used for scriptless GUI testing. However, there are several reasons we deemed these tools not suited. The tools discussed were either not maintained, not obtainable, limited flexibility to add domain specific information, or not industrially verified. From the existing tools, we did extract a number of design aspects we deem important for scriptless testing tools;

- Exploration algorithm
- Flexibility to add domain specific AUT information
- GUI state information
- Concept of state
- Failure detection

From the literature we established that TESTAR is a scriptless testing tool proven valuable for desktop and web applications but does not support scriptless testing for the mobile platforms. We have therefore opted to extend the TESTAR tool for the mobile platform, leveraging the extracted design aspects. We call this new tool mTESTAR. Simultaneously, this is our second research question; "How can we extend

TESTAR for either or both iOS and Android mobile applications in an industrial environment?". The outline of the architecture of TESTAR has stayed the same but nearly all architectural components have been changed to work for mobile applications. Additionally, the unique points of abstract action, the approach to visualization of the mobile AUT, and the failure detection component have been incorporated into mTESTAR, this is all covered in Chapter 3.

mTESTAR has been validated for an industrial application; ING Bankieren, see Chapter 4. The ING Bankieren application is used to measure and compare the performance of mTESTAR with two state of the art scriptless testing tools and scripted testing. From results we conclude that mTESTAR achieves better performance than the two state of the art scriptless testing tools. Additionally, it achieves similar performance to scripted testing. From the results it can be observed the runtime of mTESTAR is significantly longer than the scripted testing to achieve this performance, and is especially slow for the iOS platform. Note that we did not take into consideration the development time of the scripted tests. Additionally, we conclude the performance of mTESTAR should be further validated through additional tests with different settings. The case study completes the second research question as it validates mTESTAR for an industrial application.

Research question 3 focuses on testing oracles, specifically; "What types of testing oracles can be used for mobile applications and how?". The literature study in Chapter 2 introduces the types or oracles for automated testing. We conclude there is very limited work done on oracles for scriptless testing in the mobile environment. From the available work on oracles, we establish that the implicit oracle is best suited for scriptless testing as it requires little maintenance and can be generally defined for different applications. An oracle framework and a number of oracles are presented and implemented, see Chapter 3, to ensure testers can create and combine oracles. However, the oracle problem is complex and more domain specific oracles should be introduced for mTESTAR to improve its error detection further.

The main research question of this research is; "How can we apply scriptless GUI testing for mobile applications in an industrial environment?". It is our believe that mTESTAR answers this question based on the results presented. With mTESTAR we have developed a tool that outperforms two state of the art tools for scriptless GUI testing in the mobile domain and is validated on an industrial application. We envision mTESTAR to be used in combination with scriptless testing to provide maximum value for the testers. Possible directions for future work is to improve the (domain specific) oracles of mTESTAR, extend the validation of mTESTAR on different applications or with different settings, and improve the execution speed of mTESTAR for Android and iOS applications.

Bibliography

[1] "Gartner forecasts worldwide it spending to grow 9% in 2021," *Gartner*, Jul 2021. [Online]. Available: https://www.gartner.com/en/newsroom/press-releases/2021-07-14-gartner-forecasts-worldwide-it-spending-to-grow-9-percent-2021 8

[2] H. Krasner, "The cost of poor quality software in the us: A 2018 report," *Consortium for IT Software Quality, Tech. Rep*, vol. 10, 2018. 8

[3] M. Cohn, *Succeeding with agile: software development using Scrum.* Pearson Education, 2010. 8

[4] E. Alégroth, R. Feldt, and P. Kolström, "Maintenance of automated test suites in industry: An empirical study on visual gui testing," *Information and Software Technology*, vol. 73, pp. 66–80, 2016. 8, 11

[5] D. Asfaw, "Benefits of automated testing over manual testing," *International Journal of Innovative Research in Information Security*, vol. 2, no. 1, pp. 5–13, 2015. 8

[6] M. Ellims, J. Bridges, and D. C. Ince, "The economics of unit testing," *Empirical Software Engineering*, vol. 11, no. 1, pp. 5–31, 2006. 8

[7] B. Hailpern and P. Santhanam, "Software debugging, testing, and verification," *IBM Systems Journal*, vol. 41, no. 1, pp. 4–12, 2002. 8

[8] T. E. Vos, P. M. Kruse, N. Condori-Fernández, S. Bauersfeld, and J. Wegener, "Testar: Tool support for test automation at the user interface level," *International Journal of Information System Modeling and Design (IJISMD)*, vol. 6, no. 3, pp. 46–83, 2015. 9, 14

[9] S. Bauersfeld, T. E. Vos, N. Condori-Fernández, A. Bagnato, and E. Brosse, "Evaluating the testar tool in an industrial case study," in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2014, pp. 1–9. 9

[10] F. P. Ricós, P. Aho, T. Vos, I. T. Boigues, E. C. Blasco, and H. M. Martínez, "Deploying testar to enable remote testing in an industrial ci pipeline: a case-based evaluation," in *International Symposium on Leveraging Applications of Formal Methods.* Springer, 2020, pp. 543–557. 9

[11] H. Chahim, M. Duran, and T. E. Vos, "Challenging testar in an industrial setting: the rail sector." 9

[12] T. Schadler and J. C. McCarthy, "Mobile is the new face of engagement," *Forrester Research*, vol. 13, pp. 1–30, 2012. 9

[13] "Share of android os of global smartphone shipments from 1st quarter 2011 to 2nd quarter 2018." [Online]. Available: https://www.statista.com/statistics/236027/global-smartphone-os-market-share-of-android 9

[14] P. Borasi and S. Baul, *Mobile Application Market by Marketplace (Apple iOS Store, Google Play Store, and Other Marketplaces) and App Category (Gaming, Entertainment & Music, Health & Fitness, Travel & Hospitality, Retail & E-Commerce, Education & Learning and Others): Global Opportunity Analysis and Industry Forecast, 2019–2026.* Allied Market Research. 9

[15] M. Linares-Vásquez, K. Moran, and D. Poshyvanyk, "Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE, 2017, pp. 399–410. 9

[16] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE transactions on software engineering*, vol. 41, no. 5, pp. 507–525, 2014. 9, 13, 14

[17] G. Jahangirova, "Oracle problem in software testing," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 444–447. 9

[18] A. Memon, I. Banerjee, and A. Nagarajan, "What test oracle should i use for effective gui testing?" in *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.* IEEE, 2003, pp. 164–173. 9, 13, 14

[19] M. K. Sein, O. Henfridsson, S. Purao, M. Rossi, and R. Lindgren, "Action design research," *MIS quarterly*, pp. 37–56, 2011. 10

[20] Jul 2021. [Online]. Available: https://ivves.eu/ 10

[21] P. Patel, G. Srinivasan, S. Rahaman, and I. Neamtiu, "On the effectiveness of random testing for android: or how i learned to stop worrying and love the monkey," in *Proceedings of the 13th International Workshop on Automation of Software Test*, 2018, pp. 34–37. 11, 17

[22] T. E. Vos, P. Aho, F. Pastor Ricos, O. Rodriguez-Valdes, and A. Mulders, "testar–scriptless testing through graphical user interface," *Software Testing, Verification and Reliability*, vol. 31, no. 3, p. e1771, 2021. 11, 34

[23] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, A. Bertolino *et al.*, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013. 12

[24] N. Nyman, "Using monkey test tools," *Soft. Testing and Quality Eng.*, 2000. 12

[25] J. J. Holdsworth, "The nature of breadth-first search," 1999. 12

[26] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972. 12

[27] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Humanoid: A deep learning-based approach to automated black-box android app testing," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 2019, pp. 1070–1073. 12, 18

[28] V. Riccio, "Enhancing automated gui exploration techniques for android mobile applications." Ph.D. dissertation, University of Naples Federico II, Italy, 2018. 12

[29] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992. 12

[30] D. Whitley, "A genetic algorithm tutorial," *Statistics and computing*, vol. 4, no. 2, pp. 65–85, 1994. 12

[31] F. de Gier, D. Kager, S. de Gouw, and E. T. Vos, "Offline oracles for accessibility evaluation with the testar tool," in *2019 13th International Conference on Research Challenges in Information Science (RCIS).* IEEE, 2019, pp. 1–12. 13

[32] M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "A comprehensive survey of trends in oracles for software testing," *University of Sheffield, Department of Computer Science, Tech. Rep. CS-13-01*, 2013. 13

[33] L. Chen and A. Avizienis, "N-version programming: A fault-tolerance approach to reliability of software operation," in *Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8)*, vol. 1, 1978, pp. 3–9. 13

[34] S. Segura, D. Towey, Z. Q. Zhou, and T. Y. Chen, "Metamorphic testing: Testing the untestable," *IEEE Software*, vol. 37, no. 3, pp. 46–53, 2018. 13

[35] H. K. Leung and L. White, "Insights into regression testing (software testing)," in *Proceedings. Conference on Software Maintenance-1989.* IEEE, 1989, pp. 60–69. 13

[36] M. N. Irfan, C. Oriat, and R. Groz, "Model inference and testing," in *Advances in Computers.* Elsevier, 2013, vol. 89, pp. 89–139. 14

[37] O. Rodríguez-Valdés, T. E. Vos, P. Aho, and B. Marín, "30 years of automated gui testing: A bibliometric analysis," in *International Conference on the Quality of Information and Communications Technology.* Springer, 2021, pp. 473–488. 14

[38] S. R. Shahamiri, W. M. N. W. Kadir, and S. Z. Mohd-Hashim, "A comparative study on automated software test oracle methods," in *2009 fourth international conference on software engineering advances.* IEEE, 2009, pp. 140–145. 14

[39] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using gui ripping for automated testing of android applications," in *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering.* IEEE, 2012, pp. 258–261. 14

[40] Q. Xie and A. M. Memon, "Using a pilot study to derive a gui model for automated testing," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 18, no. 2, pp. 1–35, 2008. 14

[41] A. Memon, I. Banerjee, and A. Nagarajan, "Gui ripping: Reverse engineering of graphical user interfaces for testing," in *10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings.* Citeseer, 2003, pp. 260–269. 14

[42] A. Gonzalez and L. G. Reid, "Platform-independent accessibility api: Accessible document object model," in *Proceedings of the 2005 International Cross-Disciplinary Workshop on Web Accessibility (W4A)*, 2005, pp. 63–71. 15, 27

[43] S. Gojare, R. Joshi, and D. Gaigaware, "Analysis and design of selenium webdriver automation testing framework," *Procedia Computer Science*, vol. 50, pp. 341–346, 2015. 15

[44] A. I. Esparcia-Alcázar, F. Almenar, M. Martínez, U. Rueda, and T. Vos, "Q-learning strategies for action selection in the testar automated testing tool," *6th International Conferenrence on Metaheuristics and nature inspired computing (META 2016)*, pp. 130–137, 2016. 15, 26

[45] A. Valmari, "The state explosion problem," in *Advanced Course on Petri Nets.* Springer, 1996, pp. 429–528. 16, 33

[46] C. Tesoriero, *Getting started with OrientDB.* Packt Publishing Ltd, 2013. 16

[47] J. Eskonen, J. Kahles, and J. Reijonen, "Automating gui testing with image-based deep reinforcement learning," in *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, 2020, pp. 160–167. 17, 34

[48] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 94–105. 17

[49] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, 2013, pp. 641–660. 18, 34

[50] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Droidbot: a lightweight ui-guided test input generator for android," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2017, pp. 23–26. 18, 34

[51] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based gui testing of android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 245–256. 19, 34

[52] ELLA, "A tool for binary instrumentation of android apps," in *https://github.com/saswatanand/ella*. 19, 43

[53] EMMA, "open-source toolkit for measuring and reporting java code coverage," in *http://emma.sourceforge.net*. 19, 43

[54] N. P. Borges, J. Hotzkow, and A. Zeller, "Droidmate-2: a platform for android test generation," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 916–919. 19, 34

[55] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps," in *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, 2014, pp. 204–217. 19, 34

[56] N. Mirzaei, H. Bagheri, R. Mahmood, and S. Malek, "Sig-droid: Automated system input generation for android applications," in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2015, pp. 461–471. 20

[57] D. Adamo, M. K. Khan, S. Koppula, and R. Bryce, "Reinforcement learning for android gui testing," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, 2018, pp. 2–8. 20

[58] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, no. 1, 2016. 20

[59] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk, "Crashscope: A practical tool for automated testing of android applications," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2017, pp. 15–18. 21

[60] StatCounter-Global-Stats, "Mobile operating system market share worldwide," in *https://gs.statcounter.com/os-market-share/mobile/worldwide*, Jul 2021. 23

[61] M. Hans, *Appium Essentials*. Packt Publishing Ltd, 2015. 27

[62] Q. Xie and A. M. Memon, "Designing and comparing automated test oracles for gui-based software applications," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 16, no. 1, pp. 4–es, 2007. 33

[63] J. Lentz, "User interface design for the mobile web," Jul 2011. [Online]. Available: https://developer.ibm.com/articles/wa-interface/ 34

[64] R. Ramler and R. Hoschek, "How to test in sixteen languages? automation support for localization testing," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2017, pp. 542–543. 39

[65] S. Lafortune, "Discrete event systems: Modeling, observation, and control," *Annual Review of Control, Robotics, and Autonomous Systems*, vol. 2, pp. 141–159, 2019. 39

[66] K. Bogdanov and N. Walkinshaw, "Computing the structural difference between state-based models," in *2009 16th Working Conference on Reverse Engineering*. IEEE, 2009, pp. 177–186. 40

[67] A. Jalal-Karim and A. M. Hamdan, "The impact of information technology on improving banking performance matrix: Jordanian banks as case study," in *European Mediterranean and Middle Eastern Conference on Information System*, 2010, pp. 21–33. 42

[68] L. Lazic and N. Mastorakis, "Cost effective software test metrics," *WSEAS Transactions on Computers*, vol. 7, no. 6, pp. 599–619, 2008. 42

[69] A. Romdhana, M. Ceccato, G. C. Georgiu, A. Merlo, and P. Tonella, "Cosmo: Code coverage made easier for android," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2021, pp. 417–423. 43

[70] M. Rosenblatt, "A central limit theorem and a strong mixing condition," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 42, no. 1, p. 43, 1956. 44

[71] A. Ghamesi and S. Zahediasl, "Normality test for statistical analysis," *A Guide for Non-Satisticians*, 2012. 45

[72] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical software engineering*, vol. 14, no. 2, pp. 131–164, 2009. 53

[73] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012. 53

Appendix A

System information

A.1 Device for mTESTAR runs

Key	Value
Model Name	MacBook Pro
Processor Name	8-Core Intel Core i9
Processor Speed	2,3 GHz
Number of Processors	1
Total Number of Cores	8
L2 Cache (per Core)	256 KB
L3 Cache	16 MB
Hyper-Threading Technology	Enabled
Memory	32 GB
Boot ROM Version	1554.140.20.0.0 (iBridge: 18.16.14759.0.1,0)
System Version	macOS 10.15.7 (19H1323)
Kernel Version	Darwin 19.6.0
Boot Volume	Macintosh HD
Boot Mode	Normal
Secure Virtual Memory	Enabled
System Integrity Protection	Enabled

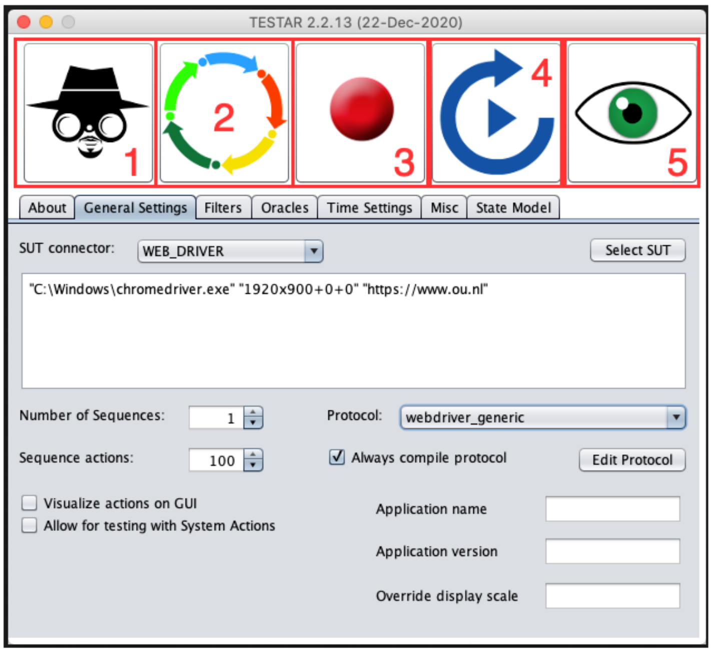
A.2 Emulator

Key	Value
CPU/ABI	Google APIs Intel Atom (x86)
Target	google_apis Google APIs (API level 30)
SD Card	512M
runtime.network.speed	full
hw.accelerometer	yes
hw.device.name	Nexus 6
hw.lcd.width	1440
hw.initialOrientation	Portrait
image.androidVersion.api	30
hw.mainKeys	no
hw.camera.front	emulated
hw.gpu.mode	auto
hw.ramSize	1536
hw.cpu.ncore	4
hw.keyboard	yes
hw.sensors.proximity	yes
hw.lcd.height	2560
vm.heapSize	384
hw.device.manufacturer	Google
hw.gps	yes
hw.camera.back	virtualscene
hw.lcd.density	560
hw.arc	false
hw.trackBall	no
hw.battery	yes
hw.sdCard	yes
runtime.network.latency	none
hw.sensors.orientation	yes
avd.ini.encoding	UTF-8
hw.gpu.enabled	yes

Appendix B

Running TESTAR

TESTAR has a GUI to set certain settings and select what TESTAR should do. In Figure B.1 the five main buttons to start different modes are highlighted. The settings available in the GUI will be explained in Section B.1. Box number one highlights the link to activate the spy mode of TESTAR. In this mode the SUT is launched and additional information about the widgets present in the GUI of the SUT are displayed. The spy mode can help to supply TESTAR with domain knowledge about the application if required. Box number two highlights TESTAR's generate mode, this is the mode in which TESTAR starts its general execution flow and tests the SUT. Box number three highlights the record mode, in here the tester can record a sequence that can be replayed by TESTAR to see if the same results are obtained. Box number four highlights the replay mode though which TESTAR can replay a sequence of actions recorded in the record mode. Lastly, box number five is the inspect mode, in this mode TESTAR allows the tester the inspect all steps of a sequence recorded in the record mode.



SUT connector: WEB_DRIVER

"C:\Windows\chromedriver.exe" "1920x900+0+0" "https://www.ou.nl"

Number of Sequences: 1

Protocol: webdriver_generic

Sequence actions: 100

☑ Always compile protocol

☐ Visualize actions on GUI

☐ Allow for testing with System Actions

Application name

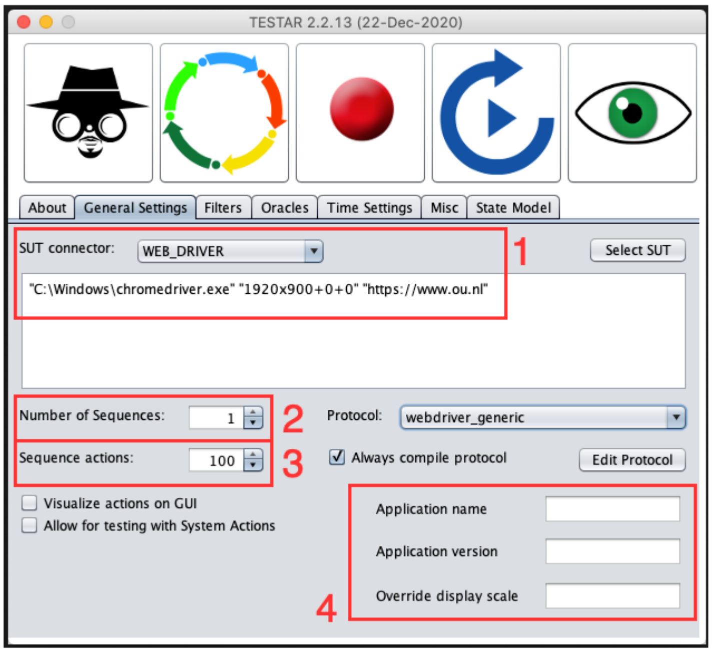
Application version

Override display scale

Figure B.1: Screenshot of the TESTAR GUI highlighting the buttons to select different TESTAR modes.

B.1 Settings

TESTAR provides a GUI in which the tester can specify certain settings concerning the runs TESTAR will perform. In this section, the general settings will be explained. Figure B.2 shows the first screen presented when launching TESTAR. In box number one, it is specified which application should be launched and tested by TESTAR. In this specific example, the target application is a website so the chromedriver.exe location and the URL of the website are specified. Box number two specifies how many sequences TESTAR should run before terminating (corresponds to previously mentioned step1 - step 6). Box number three specifies how many actions are executed in each sequence. Lastly, box four allows you to specify the name of the application and the version of the application such that the results will be stored with these textual identifiers.


TESTAR 2.2.13 (22-Dec-2020)

SUT connector: WEB_DRIVER

"C:\Windows\chromedriver.exe" "1920x900+0+0" "https://www.ou.nl"

Number of Sequences: 1

Sequence actions: 100

Protocol: webdriver_generic

☑ Always compile protocol

☐ Visualize actions on GUI

☐ Allow for testing with System Actions

Application name

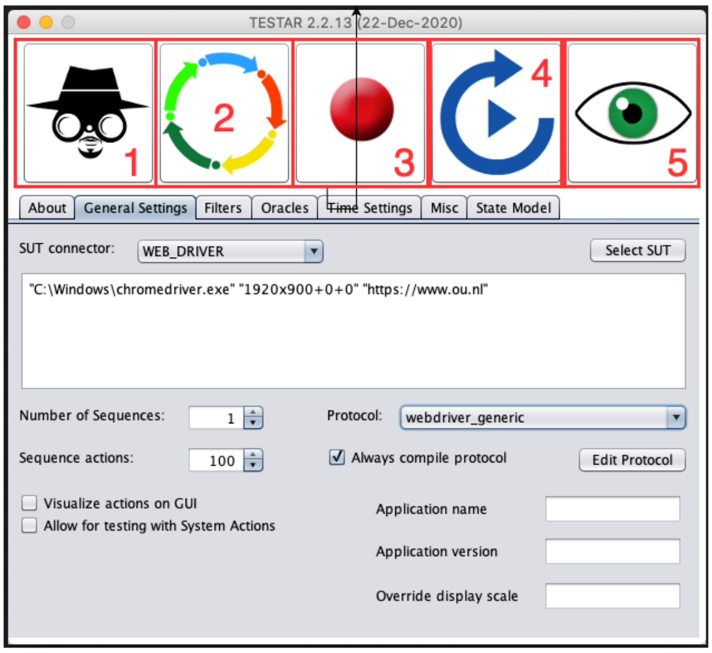
Application version

Override display scale

Figure B.2: Screenshot of the first screen presented when opening TESTAR

Figure B.3 shows a screenshot of the filter settings tab in TESTAR. In the top textbox on this screenshot, regular expressions matching certain words can be specified. If a regular expression is matched to text in a component of the GUI of the SUT, this component will not be interacted with. The second box allows for specifying regular expressions as well. If the regular expression is matched to a process running on the host machine, this process is terminated.

Figure B.4 shows the time settings available in the TESTAR GUI. Action duration refers to how much time is reserved for the execution of a single action. Action wait time specifies how long TESTAR will wait for the result of an action to be reflected to the SUT. These are both required as occasionally actions can be performed too fast for TESTAR to recognize. Startup time is the time allowed by TESTAR for the SUT to launch before starting interaction, required to make sure no interaction takes place with a still unavailable application. Lastly, max test time specifies the upper limit of a TESTAR run, even if not all sequences have been completed, if the runtime reaches this number TESTAR will quit testing.



SUT connector: WEB_DRIVER

"C:\Windows\chromedriver.exe" "1920x900+0+0" "https://www.ou.nl"

Number of Sequences: 1

Protocol: webdriver_generic

Sequence actions: 100

[✓] Always compile protocol

[ ] Visualize actions on GUI

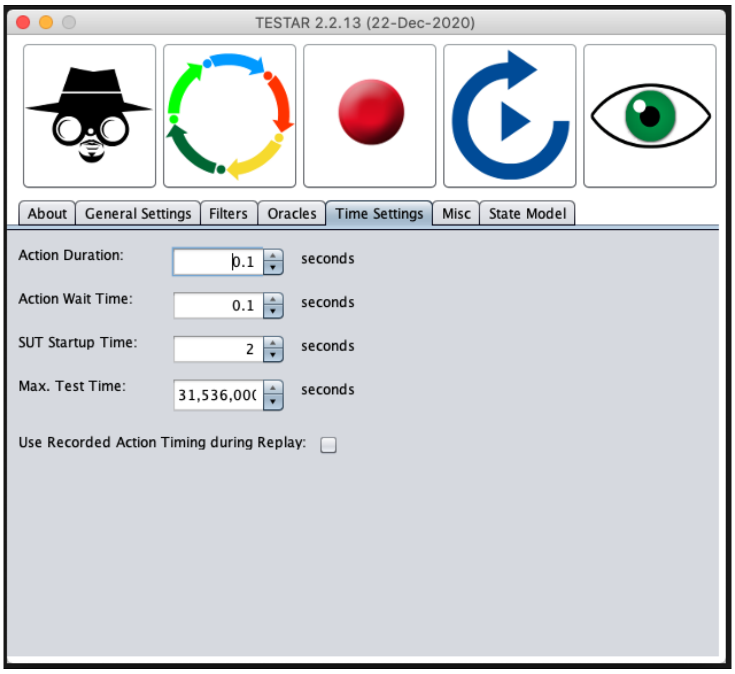
[ ] Allow for testing with System Actions

Application name

Application version

Override display scale

Figure B.3: Screenshot of the TESTAR GUI filter settings



Action Duration: 0.1 seconds

Action Wait Time: 0.1 seconds

SUT Startup Time: 2 seconds

Max. Test Time: 31,536,00(seconds

Use Recorded Action Timing during Replay: [ ]

Figure B.4: Screenshot of the TESTAR GUI time settings