Scriptless GUI Testing on Mobile Applications

Thorn Jansen[1], Fernando Pastor Ricós[2], Yaping Luo[1], Kevin van der Vlist[1],
Robbert van Dalen[1], Pekka Aho[3], and Tanja E. J. Vos[2,3]
[1]ING, The Netherlands
[2]Universitat Politècnica de València, Spain
[3]Open Universiteit, The Netherlands

***Abstract*—Traditionally, end-to-end testing of mobile apps is either performed manually or automated with test scripts. However, manual GUI testing is expensive and slow, and test scripts are fragile for GUI changes, resulting in high maintenance costs. Scriptless testing attempts to address the costs associated with GUI testing. Existing scriptless approaches for mobile testing do not seem to fit the requirements of the industry, specifically those of the** ING**. This study presents an extension to open source** TESTAR **tool to support scriptless GUI testing of Android and iOS applications. We present an initial validation of the tool on an industrial setting at the** ING**. From the validation, we determine that the extended** TESTAR **outperforms two other state-of-the-art scriptless testing tools for Android in terms of code coverage, and achieves similar performance as the scripted test automation already in use at the** ING**. Moreover, we see that the scriptless approach covers parts of the application under test that the existing test scripts did not cover, showing the complementarity of the approaches, providing more value for the testers.**

Keywords*—*Mobile apps; test automation; industrial case study

I. Introduction

Software development is a trillion dollar industry suffering from the high costs associated with software failures. Software testing is an important technique to assure quality and reduce the cost of failures. The rapidly-growing mobile application market[1] has created an urgent demand for testing techniques to assure the quality of Android and iOS applications.

Testing at the Graphical User Interface (GUI) level is a crucial part of the overall testing process. Manual GUI testing is slow and expensive [2]. Therefore, tests are often automated with scripts that can be automatically executed. Scripts reduce the need for manual GUI testing, but when the AUT changes, the scripts require maintenance effort [3]. In addition, GUIs tend to have a large state space, so creating scripts to cover all the possible paths of the AUT is usually an infeasible effort.

Scriptless testing [4] is an approach that could potentially reduce the amount of scripted test cases and the related maintenance effort. It is a process in which the tests are both generated and executed automatically on-the-fly, saving resources and time. There are, however, still a few challenges that have to be addressed to effectively apply the approach in an industrial setting.

The first challenge is the oracle problem [5], [6]. Oracles are components that distinguish whether the AUT has quality issues (failures) or meets the requirements specified [5]. The literature on oracles for automated testing is limited and almost non-existent for scriptless testing [6]. However, test oracles significantly contribute to test effectiveness and reduction of costs [7].

The second challenge is that with real applications, the user has to provide application or domain specific information to the tool to get a good coverage during automated GUI exploration. The flexibility to add domain-specific AUT information to the scriptless testing tool is an important aspect for industrial take-up.

The third challenge is related to the availability and maintenance of the tools themselves. The majority of the available scriptless GUI testing tools for Android are academic tools that are neither well maintained and established, nor properly verified on industrial-grade software applications. The situation is even worse for iOS applications. Industry take up of academic tools is more likely for tools that have open source licenses, are actively maintained, and come with case studies showing that the tool can be used in an industrial setting.

The state-of-the-art study of scriptless approaches for mobile application testing (see Section II) did not provide a suitable candidate for the context of the ING (i.e., effective creation of oracles, flexible addition of AUT-specific information, open source and maintained, demonstrated application in an industrial setting). No active open source projects for scriptless mobile GUI testing that would suit the requirements were found. Hence, the next option was extending an active open source scriptless GUI testing tool that supported desktop and Web applications but not yet mobile applications. TESTAR tool[4] was chosen since it matched the other requirements and had already been studied in industrial contexts and shown to be a valuable addition to the testing process [8], [9], [10], and some of the authors were already familiar with it.

The contributions of this paper include: 1) implementing support for testing Android and iOS applications into open source TESTAR tool, 2) evaluating performance of the extended TESTAR on an Android application of the ING and comparing the results with two other scriptless testing tools and existing scripted test automation of the ING.

The results of this paper show that our approach outperforms the other evaluated scriptless Android testing tools, and reached a comparable level of code coverage with the existing test scripts of ING. However, the combined code coverage of TESTAR and test scripts was significantly higher than with either approach alone, indicating that scriptless testing complements test scripts.

2693-9177/22/$31.00 ©2022 IEEE
DOI 10.1109/QRS57517.2022.00113

II. Scriptless GUI testing

This section describes the main characteristics and evaluates the existing tools for scriptless testing of mobile applications.

A. Action Selection Mechanisms (ASM)

The action selection mechanism (ASM), or exploration algorithm, is a core aspect of scriptless testing as it determines how the actions are selected to generate test sequences. With an ineffective ASM, a large part of the application might stay uncovered, and therefore untested, compared to a smarter or more suitable mechanism.

Random (*RND*) is the most basic ASM [11], [12]. It arbitrarily selects one action out of all possible actions in the current state. *Least executed actions* (*LEA*) [13], also called frequency-based algorithm [14], selects one of the least explored UI elements from the current state and executes an action on that element. It requires counting how often all the UI elements were clicked over all states. *Prioritize new actions* (*PNA*) [15] improves *RND* by selecting an action in the current state that was not available in the previous state.

More advanced ASMs require some sort of memory or model to remember what has been explored and which actions have been taken. A state model [4] can be used for this, but then a concept of state is required. Moreover, state abstraction is needed to allow skipping unnecessary details in the test sequence generation.

Random biased by predicted probabilities (*RBPP*) [13], [16] uses a learned or mined interaction model to predict the probability of a user choosing each possible action in the current state and then uses these probabilities as bias for random selection.

Unvisited action first (*UAF*) [17], [18] looks at the current state for actions that have not yet been visited. From these unvisited actions, it selects at random. If there are no unvisited actions in the current state, it uses a state model to obtain a sequence of actions that leads to a state with unvisited actions.

Breath-First-Search (*BFS*) or *Depth-First-Search* (*DFS*) ASMs aim at systematically exploring the AUT. In BFS all actions of the current state are executed, and the resulting 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. In DFS, each a new action is executed when it is found. This continues until reaching a state without unexecuted actions. The algorithm then proceeds to revert one state and continues. This stops when no new states are discovered.

Reinforcement Learning (*RL*) [19] fits perfectly with scriptless testing. The objective of RL is to guide an agent in the process of learning what action to take under different circumstances. Executing an action in a specific state provides the agent with a reward. The action that is optimal for each state is the action that has the highest reward. In GUI testing, higher rewards can be given, for example, to actions that cause more changes to the states [20], [21], to actions that increase state coverage and crash detection [22], to different types of actions [23], to actions that have not been executed yet [24], [25], or to actions that are most likely to be executed by users on similar applications as the AUT [16], [26].

Metaheuristic search-based optimization (*MSBO*) mimic some natural process (e.g., evolution [27] or ant colonies [28], [29]) that improves over time in an attempt to search an optimal solution. For GUI exploration this means that test sequences are generated for which the performance is measured. From the difference in performance, the optimization algorithm learns to create better test sequences.

Combinatorial (*Com*) [14] algorithms try to generate sequences to maximise the coverage of n-way event combinations. The objective is to minimise redundant execution of events and test as many unique event combinations as quickly as possible.

B. Test oracles for scriptless testing

The goal of testing is to find failures in the AUT, and the mechanism to detect them is to define test oracles [5]. Test oracles significantly contribute to test effectiveness and reduction of costs [7]. In scriptless testing, test oracles are more difficult to define, because the test sequences are generated on-the-fly, during the execution.

Test oracles for scriptless testing can be classified into online and offline oracles [30]. Online oracles are evaluated during the test execution, and can be used also as a stopping criteria for the test execution. Offline oracles are applied after the test sequences have completed running, e.g., validating the state model discovered by the test sequences. In [5], test oracles have been classified into four categories. Below we describe what each of them means for scriptless GUI testing.

Specified oracles use a specification of the AUT, defining what behaviour is acceptable and what behaviour is faulty. In GUI testing that could be, 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 failure has occurred. Specified oracles are not commonly used in scriptless testing, except when based on invariants or very specific system properties. In these cases, specified oracles can be leveraged in scriptless GUI testing and require only a little domain knowledge to be inserted into the oracle. An example of a specified offline oracle in scriptless testing, implemented by querying the inferred state model to find faults after the scriptless test execution, is checking whether a website conforms to accessibility standards [30].

Derived oracles are based on the information derived from artefacts, e.g., documentation, system executions, or other versions of the AUT. An example in GUI testing would be using inferred state models derived during scriptless testing to check the consistency of consequent versions of the same AUT [31], [18].

Implicit oracles rely on general or implicit knowledge to determine whether an application is in a faulty state or not. They can be applied to almost any AUT and require little maintenance if the AUT changes over time [5]. A common implicit online GUI test oracle is detecting when the AUT crashes. This is a simple oracle that can be found in almost all

scriptless testing tools that have a fault detection component. Similar oracles would be detecting whether the AUT becomes unresponsive, or whether there are unhandled exceptions or other warnings or error messages in the GUI, system output, or logs during testing.

Human oracles are not automated oracles but rather support systems to make it easier for humans to determine the correctness of the AUT. Human oracles are not suitable for scriptless online oracles, as it would require a human to be present and interact with the scriptless testing tool during the test execution. An example of an offline human oracle would be analysing the behaviour of the SUT by visual assessment of the inferred state models, after the scriptless test execution. That would require visualisation of the state model, for example using screenshots automatically captured during testing.

C. Domain or application-specific information

To get a good coverage during automated GUI exploration, the scriptless testing tool might need domain or application-specific information, for example username and password for a login screen.

The flexibility to add domain-specific AUT information to the scriptless testing tool is an important aspect for the industrial take-up of a tool. However, because scriptless testing aims to reduce maintenance effort compared to test scripts, adding AUT-specific information to the tool and maintaining it should be as easy as possible for the tester and not create an additional maintenance burden.

Since scriptless GUI testing tools generate the test sequences during run-time, the tools require a way to know when and where to use the provided application-specific information. One option is to use triggered behaviour, specifying a unique GUI element to recognise when, and element locators to specify where to use the pre-specified data.

D. Concept of state

Since the more advanced ASMs often use some kind of abstract state model for navigation, we also researched state abstraction for mobile AUTs. Unfortunately, many approaches, e.g., Sapienz [32], CrawlDroid [33] and DroidBot [34], do not explain in detail how they define and abstract the state of the AUT. The following state abstraction approaches have been used:

- Available actions to define the state of the AUT [23].
- Android Activity as the abstract state [35].
- The same widgets on the screen (identical hierarchy tree of widgets) [34].
- Identical hierarchy tree of widgets and identical widget attributes [4], [36], [37].
- Identical screenshots [13], [38].

Using Android Activity as the abstract state does not work well for dynamic Android apps as a lot of changes can occur within one Android activity. Identical screenshots would lead to state space explosion 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 tool evaluation, we checked whether any concept of state is supported.

E. Existing tools for mobile applications

Although scriptless GUI testing of mobile applications is not well established in the industry, there are a number of academic (Android) scriptless GUI testing tools. In this section, we look at existing tools that are open source and published in the last 5 years, so that the supported Android version is not too old. We evaluate the tools by the previously described requirements: ASMs, flexibility for adding domain or AUT-specific information, and support for test oracles for failure detection. In addition, we check whether the tool is still actively maintained.

Sapienz is a tool for search based (SB) scriptless Android testing [32]. The search objectives are to get diversified sequences with a minimum sequence length while maximising the code coverage and faults found. Sapienz allows for integrating application-specific knowledge of the AUT. A tester can add actions that should be executed under some specified conditions. Overall, the strength of Sapienz is its ability to create a diversified set of sequences that have the potential to cover the AUT extensively. Sapienz has oracles that check for application crashes. The original academic code of Sapienz is publicly available. However, since 2017 it has been developed and used internally at Facebook.

DroidBot [34] (and its extension Humanoid [16]) is a model-based scriptless Android testing tool. It builds a model of the AUT at runtime and not through static (byte)code analysis. It has several exploration methods; *DFS*, *RND*, *UAF* and *RBPP* (only Humanoid). DroidBot allows users to create additional exploration strategy (or edit the existing strategies), and offers support for injecting application-specific information of the AUT. DroidBot defined AUT states based on the GUI information and the running process information, and events based on the details of the test input and the methods/logs triggered by the input. DroidBot does not look at the correctness of the GUI states, thus lacks an oracle component. The tool is released as an open-source tool and at the time of writing DroidBot is still actively maintained. Humanoid is also open source but has not been maintained since 2019.

Stoat is a stochastic model-based scriptless Android testing tool [36]. The approach used by Stoat contains both a dynamic and static code analysis component to create a stochastic finite state machine model of the GUI of the AUT whose edges are associated with probabilities for test generation. Once the model has been created, Stoat uses a guided search algorithm, inspired by Markov Chain Monte Carlo (MCMC) sampling, to search for tests that are diverse, as well as achieve high code and model coverage. Stoat defines state of the AUT as a widget hierarchy tree, where non-leaf nodes denote layout widgets (e.g., LinearLayout) and leaf nodes executable widgets (e.g., Button). When the structure (and properties) changes, a new state is created. Stoat has no flexibility for

TABLE I
SUMMARY OF EXISTING TOOLS FOR SCRIPTLESS MOBILE TESTING.

	Sapienz [32]	**DroidBot** [34]	**Humanoid** [16]	**Stoat** [36]	**DroidMate-2** [13]	**CrawlDroid** [33]	**AutoDroid** [14]	**AimDroid** [23]
Year of publication	2016	2017	2019	2017	2018	2018	2018	2017
Actively maintained	Not OS	Yes	No	No	No	No	No	No
ASM	*MSBO*	*DFS, RND, UAF*	*RL, RBPP*	*MSBO*	*RND, LEA, RBPP*	*BFS, RL*	*RND, LEA, Com*	*BFS, RL*
Flexibility domain knowledge	Yes	Yes	No	No	Yes	No	No	No
Concept of state	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes
Failure detection	implicit (crashes)	no oracle	no oracle	implicit (crashes)	implicit (crashes)	implicit (crashes)	no oracle	implicit (crashes)

injecting application-specific information without drastically editing source code. Stoat is released as an open-source tool but has not been maintained since early 2020.

DroidMate-2 is a scriptless Android testing tool [13]. DroidMate-2 sits between the Android OS and the AUT and uses the native Android UIAutomater tool to retrieve the GUI hierarchy tree (GUI state). DroidMate-2 allows testers to develop their own action selection strategy but offers default strategies as well: RND, LEA and RBPP. DroidMate-2 generates state and widget identifiers using a selected set of attributes and widgets for abstraction, trying to disregard UI elements not belonging to the AUT. To evaluate the performance of the exploration of the GUI and to monitor for any crashes, DroidMate-2 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, but it also means instrumentation of the AUT is required. This implies either the AUT's code must be open source or open to repackaging. DroidMate-2 is a flexible tool allowing for injecting application-specific information. Unfortunately, DroidMate-2 does not verify the correctness of the GUI state, supporting only crash oracle. DroidMate-2 has not been under active open source development since 2019.

CrawlDroid [33] groups the widgets in a GUI state that it considers to be equivalent, and uses a feedback-based exploration strategy that intends to trigger actions on groups that tend to improve code coverage. BFS has been implemented for comparison. The open source project has not been maintained.

AutoDroid [14] is a tool for automatic exploration of GUI-based Android applications. It uses combinatorial algorithm to maximise the coverage of event combinations. AutoDroid defines state based on the name of the currently running activity and the set of available actions. Available actions are determined based on the widget tree. Adamo *et al.* have proposed also to use a reinforced learning algorithm [24] for determining what action should be executed in each test step for scriptless testing. Their latest tool, DeepGUIT, expands AutoDroid into Deep Q-Network-based Android application GUI testing tool [39]. However, the paper was published after the evaluation, and at the time of writing, the tool was not yet available.

AimDroid [23] is a scriptless GUI testing tool for Android, using a mix of BFS and State–action–reward–state–action (SARSA) reinforcement algorithm in the ASM. It uses available actions to represent the current state of the GUI and two level state model for ASM. It supports crash oracle, and a crash is uniquely identified by the error message and the crashing activity. It does not seem to support domain specific knowledge.

There are also other tools, but these are not open source, for example ComboDroid [40], FARLEAD-Android [41] and CrashScope [42].

UI/Application Exerciser Monkey[1] [12] is a tool built into the Android development environment and allows generating random gestures on an Android application. It is an old tool but often used as a benchmark of performance for scriptless Android testing tools. Beyond randomly sending touch events to the Android mobile device it does not possess any unique attributes. Patel *et al.* evaluated the Monkey, and after empirical analysis of 79 applications, they concluded that manual GUI testing and monkey testing achieve very similar level of code coverage. However, the costs for monkey testing are lower than the costs for manual testing [12].

SwiftMonkey[2] is a tool designed by the Zalando company for monkey testing of iOS applications. The tool can be classified as a dumb monkey testing tool as its only capability is clicking on random screen coordinates of the iOS device. SwiftMonkey seems to be the only scriptless testing tool for iOS devices.

Table I summarises the tools we have described above. DroidBot could be suited for extension. However, DroidBot is limited in the domain specific information that can be integrated. Moreover, it has no oracles. The other available scriptless Android testing tools are not active, nor tested on industrial systems, or lack in performance and intelligence. For the iOS operating system there seems to be no suitable scriptless testing tools available.

Because none of the existing scriptless mobile testing tools matched the requirements, we decided to extend TESTAR [4], an open source tool for scriptless GUI testing that supported web and desktop applications but not yet mobile applications. TESTAR was already integrated with WebDriver, and Appium uses almost the same API, so the integration was evaluated easier than extending the other tools that support only Android. In addition, TESTAR was already being evaluated at ING for testing web applications, so there were additional synergies.

After this state-of-the-art study, new related tools have been introduced, for example [43] and App Crawler[3]. Unfortunately, these tools were not available when this research was performed.

[1]https://developer.android.com/studio/test/monkey
[2]https://github.com/zalando/SwiftMonkey
[3]https://developer.android.com/studio/test/other-testing-tools/app-crawler

III. TESTAR

TESTAR[4] is an open-source tool for the scriptless testing of desktop and web applications [4]. TESTAR supports various ASMs for generating test sequences on an AUT. To enable more advanced ASMs, TESTAR supports state model inference. TESTAR allows users to define AUT-specific behaviour in AUT-specific configuration and to programmatically define any kind of test oracles, in addition to the default implicit oracles and suspicious text oracle.

The suspicious text oracle is somewhere between implicit and specified oracle. The user can specify regular expressions that 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. If TESTAR default regular expressions, e.g., error or exception, are used, then it can be classified as an implicit oracle. The same oracle functionality has been implemented also for monitoring operating system error output (e.g., unhandled exceptions) or log files.

The set of possible actions on the AUT is derived automatically during testing based on the structure of the GUI, called widget tree. TESTAR retrieves the widget tree through the accessibility application programming interface (API) for desktop applications [44] or the Selenium Webdriver [45] for web applications. TESTAR uses the widget tree as the state of the AUT, and updates the state after each executed action. Each time TESTAR updates the state of the AUT, all defined test oracles are evaluated to check whether the new state includes any failures.

TESTAR supports various ASMs and it is possible programmatically extend or edit the existing algorithms or add new ones. The existing ASMs include random, prioritise new actions (compared to previous state) [15], Q-learning algorithm that rewards exploration [25], and prioritise unvisited actions based on state model inference [15]. In addition, other reinforcement learning, ant colony optimisation [28], and evolutionary algorithms [46] have been used for improving TESTAR action selection.

To reach all parts of the AUT, scriptless testing often requires predefined AUT-specific information, for example username and password for a login screen. AUT-specific instructions can be also filtering specific actions so that TESTAR will not test them, or defining regular expressions for suspicious text oracle or log file oracle. TESTAR has two options for AUT-specific configuration: settings file and TESTAR protocol Java class. Most of the configuration can be done in settings file, but the protocol Java class allows users to programmatically overwrite or extend any default TESTAR behaviour, including test oracles and triggered actions.

The high level logical execution flow of TESTAR is displayed in Figure 1. In more detail, the steps can be described as follows:

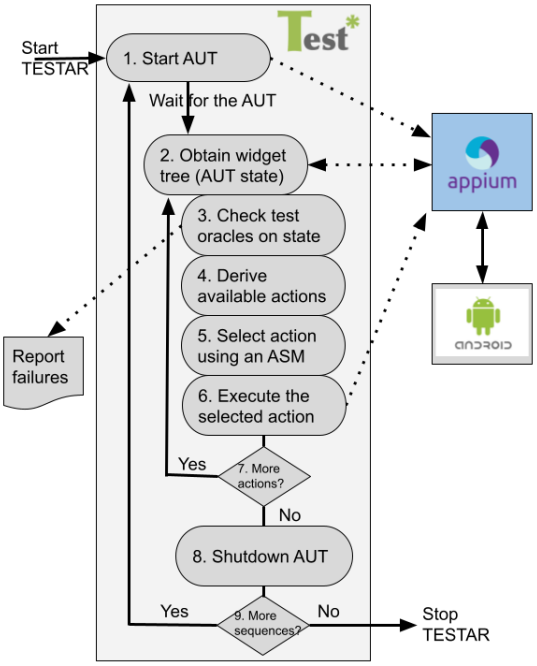

Figure 1. TESTAR logical execution flow.

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: All the defined test oracles are checked for each state. If failures are found, the report includes the sequence that found the failure.

Step 4: Determine the available actions from the obtained widget tree of the AUT.

Step 5: Use the configured ASM of TESTAR to select an action from the derived list of available actions to be executed.

Step 6: Execute the selected action on the AUT.

Step 7: If no faults are found, steps 3,4,5 and 6 are repeated until the desired sequence length has been generated.

Step 8: The resulting sequence is evaluated and the AUT is stopped.

Step 9: If the desired number of sequences has not been reached yet, the whole process is repeated from step 1. Otherwise, TESTAR finishes and exits gracefully.

IV. Extending TESTAR to support mobile testing

We have extended the previously introduced TESTAR tool with support for Android and iOS applications. Ideally, the tool should have a shared core for scriptless testing of both Android and iOS to minimise maintenance costs. Therefore, Appium[5] was chosen to be integrated into TESTAR and to be used as the API to retrieve the widget tree and execute actions

[4]https://testar.org/

[5]https://appium.io/

on the AUT. Appium is a test automation driver implementing the WebDriver API for mobile apps.

The GUI state that Appium returns is an XML document with all the Android or iOS widgets and their attributes in a tree format. This tree is parsed and each element is saved as a widget together with its corresponding attributes. Although Appium claims that it abstracts away 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 were constructed generating TESTAR state for Android and iOS separately.

Deriving available actions from the widget tree had to be implemented separately for both Android and iOS, as it 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. iOS does not support *Long-clicks* and *back* actions. Also, Appium does not support *system* actions for IOS.

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. For example, 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. For example, iOS class attribute *XCUIElementTypeButton* means the widget is clickable. Which classes should be mapped to each action may depend on the AUT, and is therefore added into TESTAR configuration.

Once a list of possible actions has been derived for a state, TESTAR uses the configured **ASM for selecting an action** that will be executed. In principle, all TESTAR ASMs should work also for the mobile extension. We will use the following three AMSs during the validation (in Section V): 1) *RND* where an action is chosen from the derived list at random, 2) *UAF* where the state model is leveraged to select an action that has not previously been executed, and 3) the reinforcement learning algorithm *Q-learning* (*QL*) that rewards GUI exploration [25] by giving higher rewards to actions that have been executed fewer times.

Although TESTAR can be configured and run from the command line interface (CLI), it provides also a GUI with a dialogue to help new users with the configuration. The GUI offers a Spy mode, a tool to manually explore the AUT in real time while being shown the attributes of the widgets in the widget tree to help with the configuration of TESTAR. The Spy mode is not used during test generation and execution. For desktop and web applications, TESTAR augments the GUI of the AUT with graphical information. Unfortunately, this approach did not work properly with mobile applications (in emulator). Therefore, a similar functionality had to be implemented by creating an extra Spy screen, showing the screenshot of the AUT and augmenting it with graphical information in this extra screen. Figure 2 illustrates the Spy screen on a mobile application. Part 1 of the Figure (indicated with a red number 1) shows the screenshot of the AUT, and the augmented green dots indicate where TESTAR recognised a click action. Part 2 of the Figure shows a selectable list of the widgets of the widget tree. Part 3 shows the attributes and attribute values of the selected widget. The Spy screen is automatically updated when the state of the AUT changes.

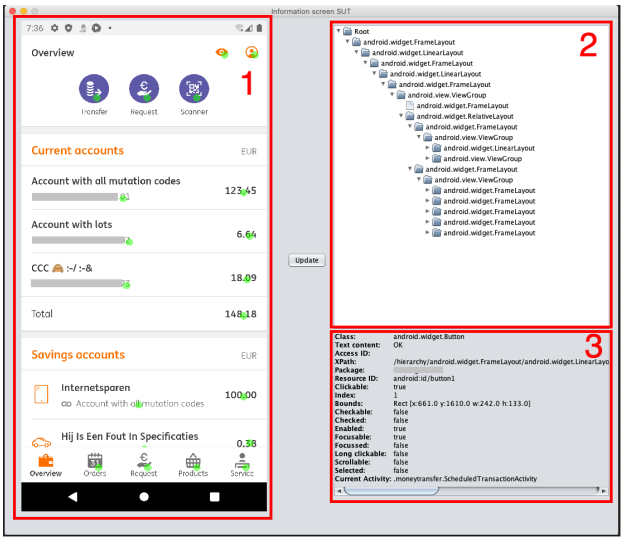

Figure 2. A screenshot of the TESTAR Spy screen for a mobile application (anonymised).

It is important to mention the difference in performance for Appium between Android and iOS. Executing operations on the AUT does not delay TESTAR in either of the operating systems. However, retrieving the state takes significantly longer for iOS because Appium retrieves all widgets of the GUI state in iOS, not only the visible ones. This leads to a massive performance hit. For Android, retrieving the GUI state takes on average 10ms on the application used in the validation. For iOS, it takes around 2000ms to retrieve the GUI state on the iOS version of the same AUT. This significantly slows down TESTAR execution for iOS applications because obtaining the GUI state is used often within TESTAR test generation flow. Although we tested also the iOS version of the ING application, the poor performance of Appium on iOS made it infeasible to get statistically significant results for the experiments on iOS. Therefore, the experiments presented in this paper were performed only on the Android version of the application.

V. Industrial Validation

Context- We performed a case study at ING , a large enterprise having a significant IT department due to the importance

Figure 3. Figure showing the top 20% of the packages with the most difference in coverage between TESTAR and Espresso.

Tool	Alg	Seqs	Acs	% IC	% LC	% MC
TESTAR	UAF	10	300	34.9%	34.5%	35.8%
TESTAR	UAF	30	100	37.2%	36.8%	38.0%
TESTAR	UAF	100	30	37.0%	37.0%	38.5%
TESTAR	UAF	60	50	34.2%	34.0%	35.4%
TESTAR	UAF	6	500	35.2%	35.1%	36.7%
TESTAR	RND	10	300	40.2%	39.9%	40.2%
TESTAR	RND	6	500	36.7%	36.7%	37.7%
TESTAR	RND	30	100	36.2%	35.6%	37.0%
TESTAR	RND	100	30	35.6%	35.5%	37.1%
TESTAR	RND	60	50	38.1%	38.3%	39.3%
TESTAR	QL	10	300	**41.0%**	**40.7%**	**40.8%**
TESTAR	QL	30	100	37.7%	37.9%	38.8%
TESTAR	QL	100	30	39.1%	39.0%	40.0%
TESTAR	QL	60	50	38.4%	38.5%	39.5%
TESTAR	QL	6	500	40.4%	40.3%	**40.8%**
Droidbot	UAF	-	-	34.9%	34.1%	35.7%
Droidbot	RND	-	-	34.3%	33.7%	35.3%
Stoat	-	-	-	23.7%	22.6%	25.7%
Espresso scripted tests				**43.9%**	**43.4%**	**45.9%**
Combined TESTAR and Espresso				**52.3%**	**52.1%**	**52.3%**

TABLE II
COVERAGE RESULTS FOR THE ANDROID APPLICATION.

of digitalization. The main goal of the company is to reduce the costs of testing - currently, a lot of manual GUI testing is still required, in addition to the test scripts. Creating and maintaining scripts to cover the whole application is too expensive.

Application Under Test (AUT) - It was in the interests of the company to evaluate the approach on their own commercial software, instead of using open source apps. The mobile application of the ING that is used as the AUT in this case study has 5 million active end users. Therefore, it fits well for validating the performance of the mobile extension of TESTAR in an industrial setting. However, due to the type of the application and the company, some details are confidential - also the details about the bugs found.

Research question - How do academic tools TESTAR, Stoat and DroidBot contribute to the effectiveness of testing when used in a real industrial environment and compared to the current automated testing practices at the ING?

Variables measured - Test effectiveness is measured through code coverage - Instruction Coverage (IC), Line Coverage (LC) and Method Coverage (MC) - using JaCoCo[6]. COSMO [47] is used to enable the integration of JaCoCo with scriptless dynamically created test sequences.

Cases of the study - Comparing the performance of scriptless testing tools TESTAR, Stoat and DroidBot, and the existing Espresso GUI test scripts developed by the Android development team of the AUT.

TESTAR is run with 15 different configurations. Three exploration algorithms supported by TESTAR: random (RND), unvisited actions (UAF) and Q-learning (QL); five combinations of *number of test sequences* (Seqs) and *number of actions per test sequence* (Acs) that all amount to 3000 actions (i.e., 6 sequences of 500 actions, 10 sequences of 300 actions, 30 sequences of 100 actions, 50 sequences of 60 actions, and 100 sequences of 30 actions.)

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 TESTAR, Stoat, and DroidBot, the number of actions for Stoat and DroidBot is also set to 3000. For TESTAR, 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.

Espresso test cases were the existing scripted test automation for the mobile application of the ING , developed by their

[6]https://www.jacoco.org/

Android developers. As Espresso tests are scripted, no settings can be modified. Although confidentiality does not allow us to give more details on the Espresso test scripts, we can reveal that these are the real set of test cases that the ING uses to test the AUT.

Procedure - All experiments have been carried out on an Android emulator running on the same device (MacBook Pro, 8-Core Intel Core i9, 2,3 GHz, 8 cores) to ensure the hardware does not affect the test results. Due to the randomness involved in the ASMs, the coverage results can differ between runs with the same settings [48], and thus, runs should be repeated many times for statistical significance. However, due to the time restrictions, it is infeasible to run each TESTAR setting combination. The average run-time was measured to be 287 minutes, meaning that 15 different settings would require approximately 90 days of continuous running. Therefore, only the best performing TESTAR settings were executed ten additional times.

The results can be found in Table II. The best performing setting for TESTAR was Q-learning with 10 sequences of 300 actions reaching a code coverage between 40.8-41.9%. This code coverage is slightly worse than the performance achieved by Espresso scripted tests. However, TESTAR achieves significantly better performance compared to Stoat and DroidBot.

As indicated, the best performing TESTAR settings was executed another 10 times for statistical significance. We found that the average of ten runs is slightly lower (IC 40.4%, LC 40.3%, MC 40.7%) than the original coverage recorded. Additionally, the variance observed is low (IC 0.29, LC 0.34, MC 0.28), indicating the randomness in the QL algorithm does not have a big impact on the overall coverage results. Together it gives us more confidence that TESTAR QL achieves performance very close to the Espresso tests.

As the performance of the Espresso tests and TESTAR are quite similar, it is interesting to examine the performance of TESTAR versus Espresso in more detail. To obtain a more detailed comparison, we investigate the code coverage at the package level. For each package, we recorded the code coverage of TESTAR and subtracted the code coverage of the Espresso tests. We found some packages showing a considerable difference between TESTAR or Espresso. The top 20% of the packages showing most code coverage difference is displayed in Figure 3. The names of the packages are anonymized as they might contain confidential information.

TESTAR achieved better code coverage for the packages containing code that implements operations and settings for the end users to modify and customise the information shown for them in the AUT. These packages contain the code relevant for customers to modify what is visible but also the code relevant to displaying the actual information to the customer. TESTAR is capable of covering this code as it is able to log-out from the application during the test sequence and continue exploring the AUT after that.

Another example of outlying packages where TESTAR outperforms Espresso in code coverage is related to exporting customer's own information and data from the application. Specifically, TESTAR covers the code related to generating PDFs with different types of information. Depending on the type of information to be exported, a different PDF generating code of the AUT is called. TESTAR seems to be capable of performing the export function from multiple contexts.

As the performance of the Espresso tests is slightly better for the complete AUT, there are numerous packages where Espresso outperforms TESTAR. Espresso achieves greater code coverage for all packages related to security. A number of the security packages are in the outliers of Figure 3. Additionally, the packages for login obtain higher code coverage through the Espresso tests. TESTAR has a predefined login sequence that is triggered in login screen of the AUT. The login was defined to always use valid credentials. This could be the cause for the lower code coverage achieved in the login packages. Adding more SUT-specific information about the login procedure to TESTAR, for example adding possibility to use wrong credentials, might increase the coverage of scriptless approach for these security packages.

Due to the difference in coverage between packages, it is interesting to determine what the code coverage is when Espresso and TESTAR are used together. Using JaCoCo, we obtain the following code coverage for Espresso and TESTAR together: IC 52.3%, LC 52.1%, MC 52.3%. This shows a significant increase in coverage. Moreover, we see that the scriptless approach covers parts of the application under test that the existing test scripts did not cover, showing the complementarity of the approaches.

VI. Discussion and threats to validity

The implemented TESTAR extension supports both Android and iOS. The other reason for concentrating on Android, in addition to poor performance of Appium on iOS, is that there is nothing to compare with on iOS. There were no test scripts nor scriptless tools available for iOS.

As described in Section V, both TESTAR, DroidBot and Stoat were supplied with the same amount of domain-specific information. Although TESTAR would allow for more, Stoat and DroidBot do not support adding more domain-specific information than provided in the experiments. This ensured that the comparison between the tools can be justified.

More test runs with all the configurations would have given more statistical confidence in the observed results. Unfortunately, that would have taken too much time from the available resources for this case study at ING. However, we did show that the randomness of the Q-learning algorithm employed was limited, and there was a low variance in the best performing TESTAR settings.

Although TESTAR and Espresso test scripts achieved quite similar code coverage results, it is complicated to fairly compare scripted and scriptless testing. For Espresso scripts, it is still relatively simple to leverage the domain knowledge of the tester by having the expected results defined as test oracles in the test scripts. This allows for testing at a functional level. For TESTAR, creating oracles with all the domain knowledge required to test at the same functional level as scripted testing

would massively increase the maintenance costs, essentially removing the advantage of scriptless testing. Instead, TESTAR is well suited to test for implicit requirement breaches. Implicit oracles require very little maintenance.

The experiments showed that TESTAR requires quite a lot of time to generate and execute the test sequences. On average, the execution time of 3000 TESTAR actions was 287 minutes on Android, whereas Espresso test scripts were executed for approximately in one hour. This difference in execution time can be an important factor affecting where in the development cycle TESTAR can be used. For nightly builds or release builds the execution time of TESTAR should not be an issue as TESTAR can be run in parallel with any other tests. However, running TESTAR every time a developer wants to commit changes may significantly slow down the development cycle. Although Espresso runs quicker, a significant amount of time is required to design and create the tests, compared to TESTAR. In addition, TESTAR supports parallel execution of using Docker containers for web application testing. This could be possible also for mobile testing and could significantly reduce the time required for test execution.

Construct validity: Code coverage was applied as a metric to evaluate the effectiveness of the testing methods. The code coverage does not directly relate to the ability of the testing tool to find faults within the software, but it is generally accepted for measuring the performance of tests.

Internal validity: We evaluated which components of the code-base should be covered by the tests together with the Android developers of the ING. This ensures the code coverage is only measured for the packages the tool should be testing. Additionally, JaCoCo is verified by manually comparing the results for identical test sequences and checking the code coverage is identical.

External validity: Although the AUT is an industrial application actively used by millions of people, it is only a single application. Additional studies are required to show that TESTAR works for other mobile applications too.

Reliability: For most of the evaluated settings of TESTAR, the tool was only executed for a few times. As the algorithms used for exploration involve randomness, the results will vary between the runs. Due to the long execution times, it was infeasible to repeat the execution of all the settings sufficiently to get statistically significant results [48]. Therefore, the reliability threat could not be completely mitigated. However, the best performing settings of TESTAR were executed an additional ten times showing low variance. This indicates that the recorded performance can consistently be achieved, reducing the reliability threat.

VII. Conclusions and future work

In this paper, we described a novel extension to the open source TESTAR tool to support scriptless testing of Android and iOS applications. The extension was validated on an Android application of the ING with millions of active end users, and compared to two other available scriptless Android testing tools (Stoat and DroidBot) and to the existing Espresso GUI testing scripts from Android development team of the application.

TESTAR outperforms both Stoat and DroidBot in terms of code coverage.When comparing TESTAR to the Espresso test scripts, the results indicate that scriptless testing complements test scripts, as TESTAR and the Espresso scripts covered different parts of the code. Consequently, the combined coverage of both approaches was higher than with either approach alone. Due to the type of the AUT, the end users expect that the application can be trusted to work flawlessly in all cases. Therefore, also the paths outside the main user scenarios are important to be covered during testing.

Overall, we believe TESTAR has value in the process of testing industrial applications, but should be used together with scripted testing. TESTAR has been used in many industrial evaluations for desktop and web applications, but this was the first time evaluating the complementarity of scripted and scriptless testing. In addition, this was the first evaluation in the mobile application domain.

In the future, we will continue evaluating the extended TESTAR on mobile applications, increasing the number of AUTs and test runs in the experiments. The evaluation at ING will continue, and the plan is to try to: 1) Find a solution for the poor performance on iOS applications, so that TESTAR could be evaluated also on iOS AUTs. 2) Develop new test oracles specifically for the AUT together with the Android developers, and use TESTAR as part of the continuous integration pipeline to evaluate its fault finding capabilities. 3) Research whether the GUI exploration algorithms can be optimised in a generic way for mobile applications. AUT-specific optimisation is possible, but might require maintenance when the AUT changes, so a more generic approach is preferred.

The TESTAR extension was implemented and evaluated at ING. The source code of the extension is in the process of being published back into the open source TESTAR project.

This collaboration was realized at the Software Engineering and Automation chapter of the OmniChannel API Platform at ING with support of the Open University of the Netherlands and Universitat Politècnica de València. Moreover, it was partly funded through the Industrial-grade Verification and Validation of Evolving Systems (IVVES) ITEA3 project[7].

References

[1] P. Borasi and S. Baul, *Mobile Application Market by Marketplace and App Category: Global Opportunity Analysis and Industry Forecast, 2019–2026*. Allied Market Research.

[2] D. Asfaw, "Benefits of automated testing over manual testing," *Int. Journal of Innovative Research in Inf. Security*, vol. 2, no. 1, 2015.

[3] R. Coppola, E. Raffero, and M. Torchiano, "Automated mobile ui test fragility: An exploratory assessment study on android," in *2nd Int. Workshop on User Interface Test Automation*, p. 11–20, ACM, 2016.

[4] 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.

[5] 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.

[7] https://ivves.eu/

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

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

[8] S. Bauersfeld, T. E. Vos, N. Condori-Fernández, A. Bagnato, and E. Brosse, "Evaluating the testar tool in an industrial case study," in *Int. Symp. on Empirical Software Eng. and Measurement*, pp. 1–9, 2014.

[9] F. P. Ricós, P. Aho, T. E. 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 *Int. Symp. on Leveraging Applications of Formal Methods*, pp. 543–557, Springer, 2020.

[10] H. Chahim, M. Duran, and T. E. Vos, "Challenging testar in an industrial setting: the rail sector," in *Information Systems Development: Designing Digitalization (ISD2018)*, 2018.

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

[12] 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*, pp. 34–37, 2018.

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

[14] D. Adamo, D. Nurmuradov, S. Piparia, and R. Bryce, "Combinatorial-based event sequence testing of android applications," *Information and Software Technology*, vol. 99, pp. 98–117, 2018.

[15] A. van der Brugge, F. P. Ricos, P. Aho, B. Marín, and T. E. Vos, "Evaluating TESTAR's effectiveness through code coverage," in *XXV JISBD* (S. Abrahão Gonzales, ed.), SISTEDES, 2021.

[16] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Humanoid: A deep learning-based approach to automated black-box android app testing," in *Int. Conf. on Automated Software Engineering*, pp. 1070–1073, IEEE, 2019.

[17] P. Aho, N. Menz, T. Räty, and I. Schieferdecker, "Automated java gui modeling for model-based testing purposes," in *Int. Conf. on Information Technology: New Generations*, pp. 268–273, 2011.

[18] P. Aho, M. Suarez, T. Kanstrén, and A. M. Memon, "Murphy tools: Utilizing extracted gui models for industrial software testing," in *Int. Conf. on Software Testing, Verification and Validation Workshops*, 2014.

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

[20] L. Mariani, M. Pezze, O. Riganelli, and M. Santoro, "Autoblacktest: Automatic black-box testing of interactive applications," in *Int. Conf. on Software Testing, Verification and Validation*, pp. 81–90, 2012.

[21] C. Degott, B. Jr., N. P., and A. Zeller, "Learning user interface element interactions," in *Int. Symp. on Software Testing and Analysis*, pp. 296–306, ACM, 2019.

[22] Y. Koroglu, A. Sen, O. Muslu, Y. Mete, C. Ulker, T. Tanriverdi, and Y. Donmez, "Qbe: Qlearning-based exploration of android applications," in *Int. Conf. on Software Testing, Verification and Validation*, pp. 105–115, IEEE Computer Society, 2018.

[23] T. Gu, C. Cao, T. Liu, C. Sun, J. Deng, X. Ma, and J. Lü, "Aimdroid: Activity-insulated multi-level automated testing for android applications," in *Int. Conf. on Software Maintenance and Evolution*, pp. 103–114, 2017.

[24] D. Adamo, M. K. Khan, S. Koppula, and R. Bryce, "Reinforcement learning for android gui testing," in *Int. Workshop on Automating Test Case Design, Selection, and Evaluation*, pp. 2–8, 2018.

[25] A. I. Esparcia-Alcázar, F. Almenar, M. Martínez, U. Rueda, and T. E. Vos, "Q-learning strategies for action selection in the testar automated testing tool," *Int. Conf. on Metaheuristics and nature inspired computing*, pp. 130–137, 2016.

[26] V. Riccio, *Enhancing Automated GUI Exploration Techniques for Android Mobile Applications.* PhD thesis, University of Naples Federico II, Italy, 2018.

[27] G. I. Laţiu, O. Creţ, and L. Văcariu, "Graphical user interface testing using evolutionary algorithms," in *2013 8th Iberian Conference on Information Systems and Technologies (CISTI)*, pp. 1–6, 2013.

[28] S. Bauersfeld, S. Wappler, and J. Wegener, "A metaheuristic approach to test sequence generation for applications with a gui," in *Proceedings of the 3rd Int. Conf. on Search Based Software Engineering*, SSBSE'11, (Berlin, Heidelberg), pp. 173–187, Springer-Verlag, 2011.

[29] S. Carino and J. H. Andrews, "Dynamically testing guis using ant colony optimization (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 138–148, Nov 2015.

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

[31] 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, pp. 3–9, 1978.

[32] 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*, pp. 94–105, 2016.

[33] Y. Cao, G. Wu, W. Chen, and J. Wei, "Crawldroid: Effective model-based gui testing of android apps," (New York, NY, USA), Association for Computing Machinery, 2018.

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

[35] 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*, pp. 641–660, 2013.

[36] 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*, pp. 245–256, 2017.

[37] 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*, pp. 204–217, 2014.

[38] J. Eskonen, J. Kahles, and J. Reijonen, "Automating gui testing with image-based deep reinforcement learning," in *Int. Conf. on Autonomic Computing and Self-Organizing Systems*, pp. 160–167, IEEE, 2020.

[39] E. Collins, A. Neto, A. Vincenzi, and J. Maldonado, "Deep reinforcement learning based android application gui testing," in *Brazilian Symposium on Software Engineering*, SBES '21, (New York, NY, USA), p. 186–194, Association for Computing Machinery, 2021.

[40] J. Wang, Y. Jiang, C. Xu, C. Cao, X. Ma, and J. Lu, "Combodroid: Generating high-quality test inputs for android apps via use case combinations," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, (New York, NY, USA), p. 469–480, Association for Computing Machinery, 2020.

[41] Y. Koroglu and A. Sen, "Functional test generation from ui test scenarios using reinforcement learning for android applications," *Software Testing, Verification and Reliability*, vol. 31, no. 3, 2021.

[42] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk, "Crashscope: A practical tool for automated testing of android applications," in *Int. Conf. on Software Engineering Companion*, pp. 15–18, IEEE, 2017.

[43] F. YazdaniBanafsheDaragh and S. Malek, "Deep gui: Black-box gui input generation with deep learning," in *36th IEEE/ACM International Conference on Automated Software Engineering*, pp. 905–916, 2021.

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

[45] 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.

[46] A. I. Esparcia, F. Almenar, T. E. J. Vos, and U. Rueda, "Using genetic programming to evolve action selection rules in traversal-based automated software testing: results obtained with the TESTAR tool," *Memetic Computing*, vol. 10, no. 3, pp. 257–265, 2018.

[47] 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)*, pp. 417–423, IEEE, 2021.

[48] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014.