UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

MARLOM JOBSOM DIAS DE OLIVEIRA

AETing: an automated exploratory testing strategy based on code evolution coverage

Recife

2020

MARLOM JOBSOM DIAS DE OLIVEIRA

AETing: an automated exploratory testing strategy based on code evolution coverage

Work presented to the Post-Graduate Program in Computer Science at the Informatics Center of the Federal University of Pernambuco as a partial requirement for obtaining a Master's degree in Computer Science.

Concentration Area: Software Engineering and Programming Languages

Advisor: Alexandre Cabral Mota

Recife

2020

Catalogação na fonte
Bibliotecária Mariana de Souza Alves CRB4-2105

O48a Oliveira, Marlom Jobsom Dias de
AETing: an automated exploratory testing strategy based on code evolution coverage / Marlom Jobsom Dias de Oliveira. – 2020.
92f., il., fig., tab.

Orientador: Alexandre Cabral Mota.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2020.
Inclui referências.

1. Engenharia de Software e Linguagens de Programação. 2. Teste de interface gráfica. 3. Teste de Android. 4. Geração de casos de testes automáticos. I. Mota, Alexandre Cabral. (orientador) II. Título.

005.1 CDD (22. ed.) UFPE-CCEN 2021-30

Marlom Jobsom Dias de Oliveira

"AETing: an automated exploratory testing strategy based on code evolution coverage"

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Aprovado em: 23/12/2020.

I dedicate this work to my family and friends who openly give me support and trust during the entire process.

ACKNOWLEDGEMENTS

Firstly and foremost I must thank God, the Almighty, for all the blessing, my health, and the will to start and finish this research. I also would like to express my deep gratitude to Professor Alexandre Mota for being my advisor, who always showed enthusiasm and support, highlighting risks and opportunities. This was very important to keep me connected to reality instead of the ideal world. He was for sure an invaluable guide and motivator of this research. I am also thankful to all Formula research group members for giving me the honor to participate in such a group composed of brilliant people and for helping me to face the challenges along the way. Special thanks to Professor Augusto Sampaio and Professor Leopoldo Teixeira for the caring and the goodwill in seeing this research becoming reality. Plus, I cannot forget to thank Cláudio Magalhães, Jacinto Reis, Filipe Arruda, and Rafaela Almeida for the immeasurable contributions that helped me in setting the bounds of my research and improving my written English. Moreover, thanks to my co-worker Eduardo Matos for putting in my mind the idea of a master degree. The beginning and ending of this would not be possible without his help and motivation. Lastly, but far to be less important, I thank my family and friends for handling the distance during the period I was dedicated to my research. I have no doubts that if I got here it is because they exist. In special to my mother for showing by the example that real strength is not made by muscles nor even true intelligence is made by titles, but these things follow the Being that decides to act out properly in the real-life gathering responsibility.

"Educação é a influência deliberada e sistemática exercida pelos maturos sobre os imaturos, através da instrução, disciplina e desenvolvinlento harmônico de todas as potencialidades do ser humano: física, social, intelectual, estética e espiritual, de acordo com sua hierarquia essencial, por e para seus usos individuais e sociais, dirigidas no sentido da união do educando com seu Criador como fim último." (REDDEN; RYAN, 1973)

ABSTRACT

Testers face challenges to have their test suites up-to-date with respect to the application source code evolution to be tested. These challenges are greater in a global distributed software development context. The growing daily testing demand also makes it difficult to maintain such suites. These test suites are also continuously automated to save the time of human testers, but they require maintenance as well. Exploratory testing comes as a trade-off between test case maintenance, human expertise, and flexibility. Unfortunately, it is a manual task in general. In this work, we specifically designed a strategy called AETing. AETing receives *Test Scenarios* generated by ArcWizard, which are the results of code static analysis that produces screen navigations suggestion that covers code change, and AETing maps such static screen navigations suggestion into concrete code implementation based on Page Objects maintained by Motorola in its testing framework called Page Browser. AETing combines these two resources to generate automated test cases that perform screen navigations and Monkey testing aiming to maximize coverage of code changes evolution between two versions of a given Android application. We developed and evaluated our approach in a real testing operation environment related to Motorola Mobility, through a partnership between CIn-UFPE and this company supported by the Informatics Law. The evaluation consisted of testing four different Motorola Android applications. Through the evaluation, we obtained promising results concerning the comparison between AETing and expert exploratory testers' code coverage. We discuss in detail how AETing works and the results achieved.

Keywords: GUI testing. Android testing. Automated test case generation. Automatic verification. Monkey.

RESUMO

Testadores enfrentam desafios para ter suas suítes de testes atualizadas em relação à evolução do código fonte da aplicação a ser testada. Estes desafios são maiores dentro de um contexto de desenvolvimento de software globalmente distribuído. A crescente demanda diária por testes também torna difícil a manutenção de tais suítes. Estas suítes de testes são, também, continuamente automatizadas para economizar tempo dos testadores humanos, mas elas também requerem manutenção. O teste exploratório vem como um balanceamento entre manutenção de casos de testes e a experiência humana e flexibilidade. Infelizmente, em geral, esta é uma atividade manual. Neste trabalho nós desenvolvemos especificamente uma estratégia chamada AETing. AETing recebe *Test Scenarios* gerados por ArcWizard, que são resultados de análise estática de código que produz sugestões de navegação de tela que cobrem mudança de código, e AETing mapeia tais sugestões estáticas de navegações de tela em implementação concreta de código baseado em *Page Objects* mantidos pela Motorola em seu framework de teste chamado Page Browser. AETing comina estes dois recursos para gerar casos de testes automatizados que executam navegações de tela e teste de Monkey objetivando maximizar a cobertura da evolução da mudança de código entre duas versões de uma dada aplicação Android. Nós desenvolvemos e avaliamos nossa abordagem em um ambiente operacional real de teste relacionado à Motorola Mobility, através de um convênio entre o CIn-UFPE e esta empresa apoiado pela Lei de Informática. A avaliação consistiu em testar quatro diferentes aplicações Android da Motorola. Através da avaliação nós obtivemos resultados promissores relativos à comparação entre a cobetura de código alcançada por AETing e testadores exploratórios experientes. Nós discutimos em detalhes como AETing funciona e os resultados alcançados.

Palavras-chaves: Teste de interface gráfica. Teste de Android. Geração de casos de testes automáticos. Verificação automática. Monkey.

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Android developers and testers aim at creating Android applications (APK) with high quality. However, it is still very difficult to have test cases updated according to changes in source code (Mirzaaghaei; Pastore; Pezze, 2012). Thus, as time goes on, these test cases become somewhat obsolete (suffer a loss in coverage, for instance) since maintenance is an expensive task. Although from 20% to 50% of the software development process cost is about Verification and Validation (V&V), among V&V activities, maintenance still represents more than 60% of the effort dedicated to the testing process (ALÉGROTH; FELDT; KOLSTRÖM, 2016). That is, half of the software development process cost is related to V&V, and, in V&V, just maintenance is the most expensive activity. Additionally, in a globally distributed working scenario, where developer and tester teams are geographically dispersed, the communication delay (Herbsleb; Mockus, 2003) amplifies this problem.

Testers automate test cases to speed up productivity. However, this also increases the maintenance responsibility because there is a chain of artifacts to be synchronized: (1) the APK source code (SUT — System Under Testing), (2) the test case descriptions, and (3) the automated test cases. Beyond that, as Graphical User Interface (GUI) elements or screen navigations change, the automated test cases frequently fail, not because of SUT bugs but due to the fact that the test cases cannot handle the APK source code evolution. Exploratory testing comes as a trade-off related to human expertise and flexibility. Unfortunately, it is a manual task in general.

Therefore, we intend to develop a testing approach that would reduce the need for the maintainability of the testing artifacts by humans and deliver updated test cases that follow the APK source code evolution.

1.1 CONTEXTUALIZATION

Our work is part of a real industrial testing operation environment within Motorola Mobility, in the context of a partnership between CIn-UFPE and this company supported by the Informatics Law. Our concern is developing a proposal that would fit such a production environment. Thus, it raised the need of starting from a solid ground to deliver a robust solution into our context. Still, there is a research group where other academic initiatives are being proposed to improve the Motorola testing process in several fronts under this partnership, and this particular work is one of them.

In Academia, there are several solutions aiming at exercising a given APK without human intervention based on different techniques. Below we list some sophisticated approaches that we evaluated in the attempt to find a baseline infrastructure where our research work would start from.

JPF-Android is a model-based checking approach that extends the Java Pathfinder (JPF) model checker (Java Pathfinder Documentation, 2020) to enable the automatic verification of APKs on the standard Java Virtual Machine (JVM) (MERWE; MERWE; VISSER, 2012). It does not need a physical or virtual Android device as a platform to check a given APK. JPF-Android provides a simplified model of the Android framework based on JPF extensions that allow JPF to run the APK out of its natural environment. The main limitation of this approach relies on the model provided of the Android framework. It covers an old and small set of the Android platform (JPF-Android Documentation, 2017) and this model requires manual maintenance as well.

JPF-Mobile is another model-based checking approach that adapts JPF model checker (KOHAN et al., 2017) to run in the Android platform. It delivers an APK that wraps an adapted version of JPF to the Android environment. This way, JPF-Mobile tries to overcome the limitation of JPF-Android regarding the manual Android environment modeling. However, it can only apply model-checking features to non-GUI elements of APKs. Hence, it cannot be applied to a regular APK where the GUI is the main interface to the users.

Concolic Android TEster (CATE) is a model-based checking approach that also attempts to deliver JPF capabilities to the Android platform through concolic static analysis on APK source files (McAfee; Wiem Mkaouer; Krutz, 2017). It disassembles the APK target to re-build it as a JAR file. This JAR holds a regular JUnit test function that has all the supported APK functions that the Android framework or the user would trigger. Roboletric [1], a unit testing framework for Android, carries the simulation of the Android environment inside a JVM and operates the JUnit test function of the JAR built by CATE. It is worth noting that, similar to JPF-Android, CATE also tries to run the APK without the need for an Android device (physical or virtual). We tried to execute CATE over several APKs. Unfortunately, our trials result in failures during the re-compiling process.

EHBDroid is an event handler-based testing approach, unlike the others aforementioned

[1] https://github.com/robolectric/robolectric/

(SONG; QIAN; HUANG, 2017). It delivers a mechanism that takes advantage of the event-driven system characteristic of the Android platform. In event-driven systems there exists a relation between an event and its handler (events that occur on the GUI are sent to their correspondent handler functions, for instance). To generate all sorts of GUI events is a difficult task. Plus, some of these events require a specific system state condition to be triggered. EHBDroid interacts directly with the event handler of the GUI event through source code instrumentation instead of generating GUI inputs. We used EHBDroid for a set of APKs to evaluate its effectiveness, but during our runs, the APKs that opened did not show any change in its behavior or another indication that the GUI event handlers were being triggered. Other APKs simply failed during the parse of the `resources.arsc` files, which happens during the instrumentation phase.

DroidWalker is a model-based approach focused on reproducibility (HU; MA; HUANG, 2017). It can build a dynamic-adaptive model while it automatically controls the Android device to perform screen navigations and register the system state of such navigations as well. Once the dynamic navigation model is complete, developers or testers can select a given state from the model built and DroidWalker would drive the Android device to the selected point, ensuring the exact same state as registered during its exploration. To start its screen navigations, DroidWalker creates a testing APK from the source code of the target APK injecting a testing class responsible for manipulating the Android device, it collects the state data while browsing the target APK and sends the state data to an external desktop web server. In parallel, the desktop web server is started to receive the state data and keep building the navigation model dynamically. We also used DroidWalker for a set of APKs to evaluate the dynamic navigation model that it builds. It also failed right at the beginning of the process when generating the testing APK.

The evaluations of the aforementioned approaches were performed using the APKs they also provided. They did not behave as we expected. Consequently, we did not move a step forward in the evaluation of submitting Motorola APKs, required by our industrial context. Additionally, we also tried to contact the authors of the failed approaches to share our findings to have any support. JPF-Android's author, unfortunately, did not answer us. One of CATE's authors replied to us sharing that the students that created the solution are no longer at the university since they concluded their course. The author of EHBDroid suggested we use the tool submitting an APK based on an old Android SDK because the issue we faced may be associated with the Android version (to support newer Android versions is a requirement of our industrial context). DroidWalker's author later replied to us sharing that this tool became another one. Due to the difficulties of reproducing and using the approaches, and for the lacking of available information, on this occasion, we decided that we needed an extra solution.

1.2 MOTIVATION

The daily testing activity continues to be expansive and the synchronization of APK source code, test case description, and automated test cases is still a hard task that requires the interaction of multiple teams globally distributed. We intended to deliver a solution that would reduce the impact of the maintenance in automated testing cycles. This solution must be fully automated, not requiring any human intervention, and it must be robust enough to support the Android platform and APKs versions source code evolution. We evaluated the feasibility of reusing one of the following approaches in our context as a baseline infrastructure: JPF-Android, JPF-Mobile, CATE, EHBDroid, and DroidWalker. None of these approaches could provide a solid platform. JPF-Android does not provide a rich set of models of the Android platform and it still requires manual Android platform modeling. JPF-Mobile does only support non-GUI APKs. CATE simply did not work when we executed it, failing during the re-compiling process. EHBDroid failed during the instrumentation phase. DroidWalker failed in its initial stages trying to generate the testing APK. Beyond that, these approaches do not support up-to-date Android versions, which is crucial in our industrial context, since we are included in the testing environment of Motorola. Fortunately, we had access to use an existing testing framework from Motorola itself called Page Browser. We faced this as an opportunity to implement a new testing solution that better fits our industrial environment needs based on the available proprietary technology. Nevertheless, the idea we are proposing is not technology-dependent, and it would fit any other context as long as similar approaches are available providing mechanisms to control the device, like Page Browser does.

1.3 PROPOSAL

We developed a strategy and tool called AETing (Automatic Exploratory Testing), which automates test scenarios provided by another work from our research group called ArcWizard (REIS; MOTA, 2018). These test scenarios are created from a static analysis process identifying the code evolution between two different versions of a given APK. Test scenarios are screen navigation suggestions that cover the code change detected during the analysis. AETing generates automated test cases that cover the differences detected by ArcWizard through screen navigation and Monkey testing (Monkey Documentation, 2019). This approach does not require a human for (1) maintaining test case descriptions (2) or even automating test cases based on incoming changes. That is, the automated test cases generated by AETing automatically can be completely thrown away without any loss to the test campaign because they just depend on the availability of a new pair of versions of an APK to generate a new test suite.

Our idea consists in combining two sources of screen navigation knowledge to generate automated test cases that cover code changes. The first source is the already introduced ArcWizard test scenarios. The second source comes from the Page Browser framework resources, which follows the PageObject pattern (FOWLER, 2013). A page object is an object-oriented class that maps a GUI to allow the manipulation and interaction with its widgets through API calls. This framework holds page objects that map APK screens and their relation in terms of navigation. The page objects are implemented and maintained by Motorola developers.

The strategy is to use ArcWizard test scenarios, which point to the regions that must be covered during testing, to trace routes over Page Browser page objects, our screen navigation knowledge map. AETing maps the ArcWizard test scenarios into real implementations of the navigation suggestions in the Page Browser framework. For this, we model the Page Browser resources in a CSP_M specification, a process algebra based on events that also allows the modeling of components that interact with each other (SCHNEIDER, 1999). This model is our screen navigational map. From such a model, we use the FDR (GIBSON-ROBINSON et al., 2014) refinement tool to find the best route among the Page Browser resources available that were modeled into CSP_M from the ArcWizard test scenarios suggestions.

1.4 CONTRIBUTIONS

The main contributions of this work are:

- Creating a CSP_M specification from available page objects dynamically;
- Creating a page objects based navigation from a counter-example of a CSP_M refinement assertion;
- Creating a mechanism to generate automated test cases that match static code analysis outcome into concrete script implementation;

- Performing several experiments to investigate the benefits of using AETing in our industrial partner environment.

1.5 DISSERTATION STRUCTURE

The chapters of this work are structured as follows.

- Chapter 2 (Background): introduces the concepts and technologies upon which AETing was developed.
- Chapter 3 (Strategy): explains how AETing provides a solution of generating automated test cases automatically combining the available resources.
- Chapter 4 (AETings' Evaluations): describes an empirical study of AETing, discuss the results achieved, how AETing performed during the evaluation and its limitations.
- Chapter 5 (Conclusion): ends this work with our conclusions with respect to the tool developed, lists related work from the academic community and provides future work.

2 BACKGROUND

In this chapter we present the concepts and technologies that mold the basis of our work. We introduce concepts about *Software testing* and *Model-based testing* and we present the set of technologies upon which we developed AETing: *ArcWizard*, *Page Browser*, *CSP* and *FDR*.

2.1 SOFTWARE TESTING

Software Engineering is one of the engineering disciplines that focus on all aspects of software production. It describes a systematic sequence of activities, also known as the software process, that guides the development of this kind of product (software). The software process is embodied by four common base activities: *Specification*, when it is defined the software that will be produced and its restrictions; *Development*, when the software is designed and implemented; *Validation*, when the software is verified to ensure it accomplishes the specification; *Evolution*, when the software is updated to resonate changes that the specification may suffer (SOMMERVILLE, 2010).

The Validation common base activity is also named as verification and validation (V&V). The term verification is related to a set of actions to ensure the software attends to its specifications and the term validation to ensure that the software produced attends to the user's expectations (SOMMERVILLE, 2010). In V&V, *Software Testing* plays an important role in software quality control and bug detection still in earlier development stages. Software testing is executed in different levels during the development stages and these levels contrast to each other based on their object of testing, known as *target*, or based on the objective (BOURQUE; FAIRLEY, 2014). Three conventional testing levels are:

- **Unit Testing**: it dedicates attention to verify the unit of a software project, such as methods or object classes. This kind of test is usually performed by method calls with different parameters in isolation of other software components. Naturally, unit testing is normally applied when the source code is accessible. It is generally implemented by the programmer of the source code being tested;

- **Integration Testing**: it focuses on verifying the interfaces of the software components tested in unit. The interfaces of those components are tested through the interactions between them. It is an architecture-driven technique to build a software architecture from the incremental integration between the software components. It is also an activity that happens along the development stages;

- **System or Validation Testing**: it concerns verifying the software behavior as a complete system and it happens after the success of the other two testing layers (unit and integration). Since the internal behaviors are covered by unit and integration testing strategies, system testing directs the attention to actions and outputs visible by the users.

These three levels of testing are classified into two major testing strategies: *Black-box* and *White-box*. Black-box testing describes a strategy where the source code is not visible (e.g: system testing). Thus, this strategy relies on the input and output without knowing the internal implementation. It focuses on finding circumstances in which the software will not behave as expected based on the knowledge acquired from artifacts related to the software: the specifications, for instance. White-box testing, complementarily, describes a strategy where the source code is visible, which allows the test to be derived from the internal implementation logic of the target software (e.g: unit testing and integration testing) (MYERS; SANDLER; BADGETT, 2011). Figure 1 illustrates the hierarchy of the testing strategies presented.

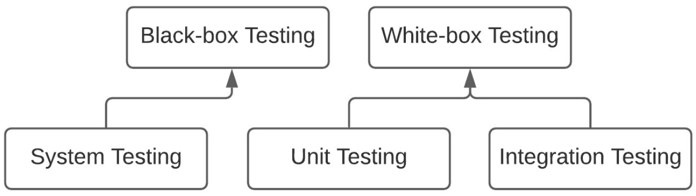


Figure 1 – Hierarchy of testing strategies

The process of generating test cases is an activity related to *Test Case Design*. As we mentioned before, the strategy adopted distinguishes from others based on its target and objective. Nevertheless, the design of a test case is also affected by the testing strategy. To design a unit test case it is required to analyze the source code of the method or object class to be verified. To design an integration test case it is necessary to know the target software architecture and the external interfaces of the components through which the interactions can happen. The design of a system test case involves higher-level artifacts such as, but not only, the specifications. Figure 2 illustrates the evolution of the visibility level. Looking at this figure from left to right, it is worth noting the rise of the visibility level: unit, architecture and use case.

The strategy that comes closest to the user experience usage of a software is the system testing strategy, and a system test case is very difficult to design and create because it has to take into account the software as a whole (MYERS; SANDLER; BADGETT, 2011). However, the Model-Based testing technique can be used on system testing level. It can reduce the dependency analysis when generating test cases for system-level (Korel; Tahat; Vaysburg, 2002).

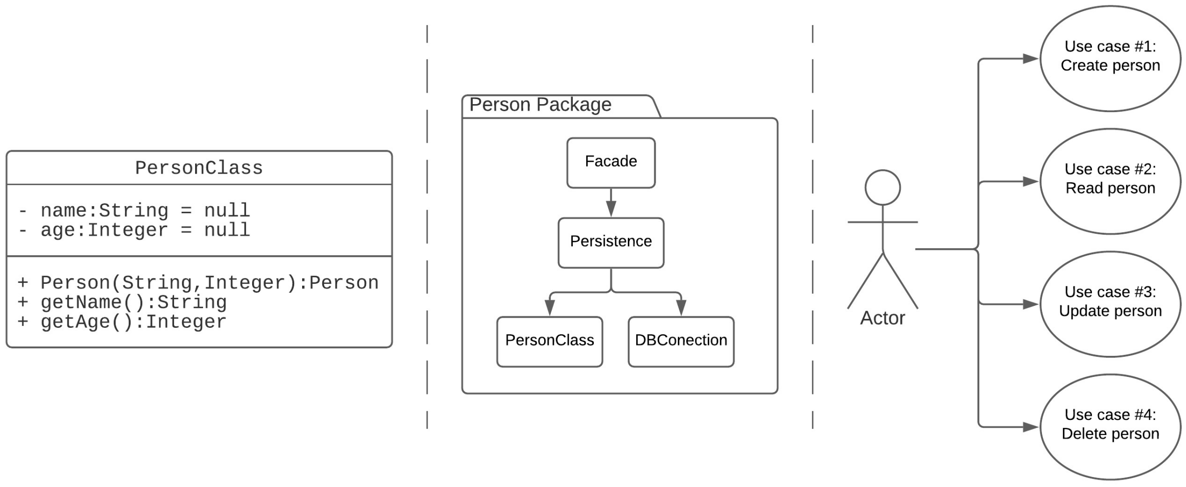


Figure 2 – Software visibility level (unit, architecture, use case diagram).

2.1.1 Model-based Testing

Model-based Testing is a Black-box strategy and it applies to the system testing level. Model-based test cases can be derived from a formal representation, or model, of the target software. The software model can be used to automatically generate test cases focused on the behavior of the target software. An example of an artifact used for this strategy may be an UML diagram, which can be part of a software specification. The elements of model-based testing include: the notation used to model the software, the algorithm used to generate the test cases automatically and the infrastructure for the test execution (BOURQUE; FAIRLEY, 2014).

We illustrate in Figure 3 a generic process of a model-based testing mechanism. It requires five major steps (PRESSMAN, 2004):

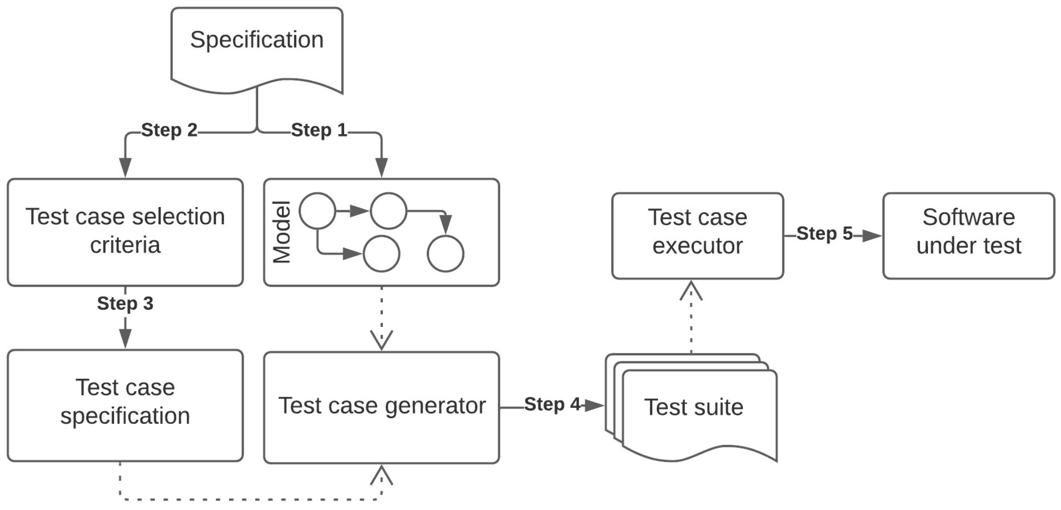


Figure 3 – Generic model-based testing process.

- **Step 1**: It analyzes or builds a formal behavioral model of the target software. This model captures how the software would respond to external events or interactions. The construction of such a model is based on specifications or other existing formal artifacts as a knowledge source. Usually, the specification is the main source. When it does not exist, or it is not accessible, the source code, developers or testers knowledge may become an acceptable alternative source;

- **Step 2**: It is defined as the criteria used to select a subset of the software model, which will represent a test case selection. This criterion is an element that contributes to extracting from the model an effective test case that is likely to find bugs. Furthermore, the criterion makes it possible to select a finite and feasible number of test cases. Good criteria would be based on the specification or in the structure of the software model;

- **Step 3**: Once we have the test case selection criteria, it is formalized into a test case specification. This specification makes it possible to engage the test case generation through an automatic mechanism (e.g: a test suite generator) to generate an operational test suite;

- **Step 4**: It is the concrete implementation of the operational test suite from the test case specification and the software model. A test case generated is selected to be further executed;

- **Step 5**: The concrete implementation of the test suite is now executed against the target software, which includes two stages: (1) the execution means the submission of a real input of the test case to the target software; (2) the building of a verdict to indicate the test case outcome. It *passes* when the expected output is achieved, it *fails* when the expected output is not achieved and it is *inconclusive* when this decision cannot be made.

2.2 ARCWIZARD

ArcWizard is a tool to aid exploratory testers to focus their work on the most recent changes detected from two different versions (code evolution) of a given APK (REIS; MOTA, 2018). The goal of ArcWizard is to provide English-based test scenarios so exploratory testers can understand and execute. ArcWizard uses source code to develop its task. Following we describe each ArcWizard's task (Figure 4) and output.

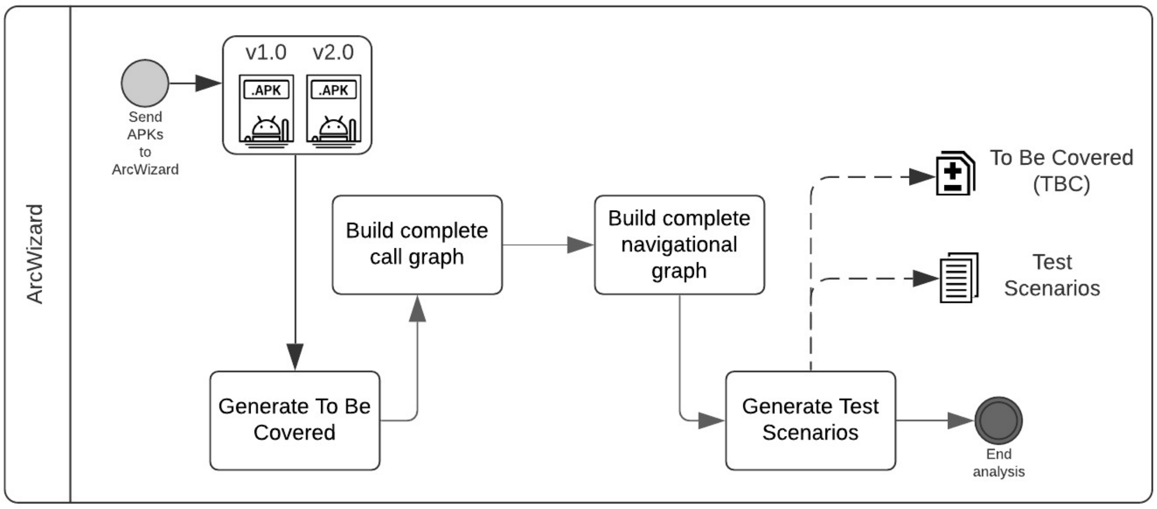


Figure 4 – ArcWizards' tasks overview.

2.2.1 To Be Covered

ArcWizard's first task is to identify code changes between two different versions of the same given APK, an old and new version. The old version of the APK is used in this stage as a reference for this difference calculation mechanism. The new APK version is named as *target*, since it is the one to be exercised during testing. That is, we want to know what to cover in this new target APK. To calculate the difference between the two APK versions, ArcWizard uses Soot (VALLEE-RAI et al., 1999), a framework that allows performing static code analysis on Java and Android applications. In this initial stage, Soot is used to retrieve from both APK versions the classes and methods to store them into two separate sets. Since an APK holds its own source code and also libraries and other assets, ArcWizard filters from those two sets only the classes and methods related to the APK itself based on the Java class package. With both sets holding the data regarding the APK, ArcWizard compares them to build a final one containing the classes and methods that were added and changed in the target APK. This final set is called *To Be Covered* (TBC).

2.2.2 Complete call graph

ArcWizard's second task is to build a call graph from the target APK. A call graph is a useful representation for inter-procedural communication between subroutines, or method calls. It helps in static analysis activities transforming the dynamic relation between method calls into a directed graph (Ryder, 1979). In a call graph, each node represents a method and the edge is the call to the next method from the previous one. An example would be `method1` $\rightarrow$ `method2`. It is important to mention that call graphs are static approximate representations of the runtime method calls' relation (GROVE; CHAMBERS, 2001). That is, it may represent correlations that are not possible to reach in runtime. ArcWizard interacts with Soot again to retrieve from the target APK the classes and methods to build a complete call graph that presents all the method call chains. The size of the call graph built depends on the size of the APK. However, ArcWizard considers only the possible relationship of the method calls that participate in the TBC, as we will see later in Section 2.2.4. Figure 5 illustrates a call graph.

2.2.3 Complete navigational graph

ArcWizard's third task is to build a navigational graph directly from the static analysis of the target APK resources. This navigational graph is represented by a cyclic directed graph. It is cyclic because the screen navigation is an action of double track, as we will see next. However, instead of the nodes being methods and the edges being directed calls, a node is an Android Activity and an edge is the combination of two things: (1) the event from the user interaction and (2) the GUI element target of the event. These events are gestures the

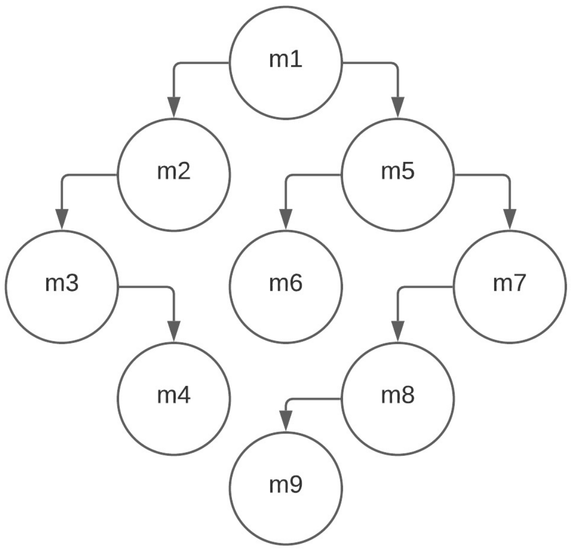


Figure 5 – Sample of a call graph.

user performs to interact with the GUI elements on the activity (an APK screen) such as clicks, long-clicks, and drag-and-drops. The GUI elements can be text fields, buttons, and date pickers, for instance. ArcWizard decompiles the target APK to retrieve the XML files which hold the GUI layouts and their IDs. A layout is associated with an activity through the method call `setContentView(layoutId)`. To find this association, the complete call graph previously built (Section 2.2.2) is used to track the `setContentView` starting from the activity lifecycle methods. Next, the layout itself is analyzed to identify its GUI elements and the events they can receive from the user interaction. To identify the relationship between activities, the complete call graph is used again to track the `startActivity` and `startActivityForResult`, methods that open other activities, starting from the event callbacks, methods that react to the user interaction. After this processing, ArcWizard builds the directed graph connecting all the activities, events, and GUI elements to compose the complete navigational graph. Similar to the complete call graph, the size of the navigational graph depends on the size of the APK. It is not a problem for ArcWizard because the complete navigational graph will be pruned based on the TBC, as we will describe in Section 2.2.4. Figure 6 exemplifies an APK with a *Main* activity that has a button that opens the *About* activity when it is clicked. In the *About* activity we have a back button that goes back to *Main*.

An Android activity is a Java class that models a screen of the APK that the user can interact with (Android Activity Documentation, 2020). It has a lifecycle that conducts its behavior and state while the user experiences the APK. For example, when the user opens an APK it

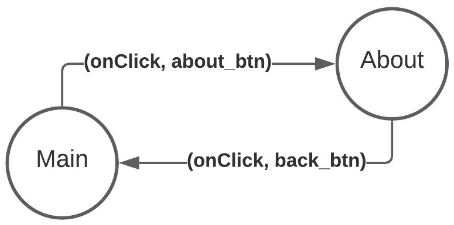


Figure 6 – Navigational graph of an hypothetical APK

raises a screen and calls the following chain of methods `onCreate` → `onStart` → `onResume`. After this, the screen will be visible and accessible to the user. If a notification pop-up appears over the top of the current screen, the onPause method is triggered. When the user goes to another screen and the previous one is not visible, the chain `onStop` → `onDestroy` may be executed to completely close the screen. We say *may* because there is a chance of the not visible screen to be reopened. In this case, the chain called will be `onStop` → `onRestart` → `onStart` → `onResume`. Figure 7 presents an overview over the Activity lifecycle.

Beyond the lifecycle, an activity can also be defined as the entry-point of an APK. This activity is commonly named Main. It is the place where the user starts the usage experience from. The activity has GUI elements that expect user interaction. Each GUI event is related to a corresponding *Listener* (Android Input Documentation, 2020), a mechanism that observes the user interactions on a GUI element and triggers the proper callback method (the reaction). For instance, a click is observed by the `OnClickListener` for a button and it triggers the `onClick` callback when the user clicks on the button.

In summary, the user interaction exercises the APK inner methods that compose the Activity lifecycle and the callbacks related to the GUI elements in the Activity as well. That is, the user can exercise an APK browsing over the screens and also interacting with their GUI elements by clicking, scrolling, dragging, focusing, and all other events supported by the Android platform.

2.2.4 Test Scenarios

ArcWizard's fourth and last task is to prune the complete navigational graph built based on the TBC. The pruned navigational graph will keep only the screen navigations that can reach those methods that should be covered during a testing execution, the methods listed in

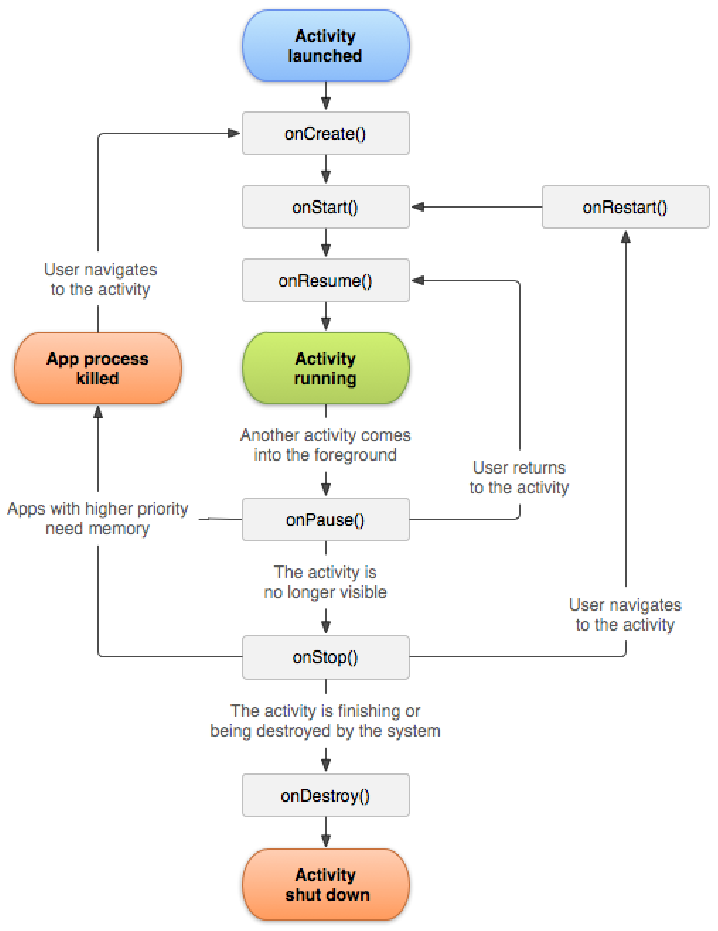

Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Figure 7 – Diagram of the state paths of an Activity. (Android Activity Documentation, 2020)

the TBC. With the TBC set and the complete call graph of the APK, ArcWizard calculates reachable paths from the TBC to GUI elements. And from the set of GUI elements, ArcWizard creates a pruned navigational graph, from which test scenarios are extracted. Figure 8 illustrates the generation of the pruned navigational graph. The illustration shows, from the left to the right, a TBC, a pruned call graph, and a pruned navigational graph. The transparent elements represent those ignored by ArcWizard due to the prune done based on the TBC. The pruned call graph has yellow nodes, methods listed in the TBC. From these yellow nodes, it is traced upper methods calls that are related to a GUI element. The node in green is the method `onClick`, which is related to the *Main* activity in the pruned navigational graph. The pruned navigational graph holds the *Main* and *About* activities keeping the navigation that exercises the `onClick` method, which covers the TBC.

The nodes of the pruned navigational graph are the screens and the edges are methods along with the GUI element that belongs to the source screen. The nodes may have multiple

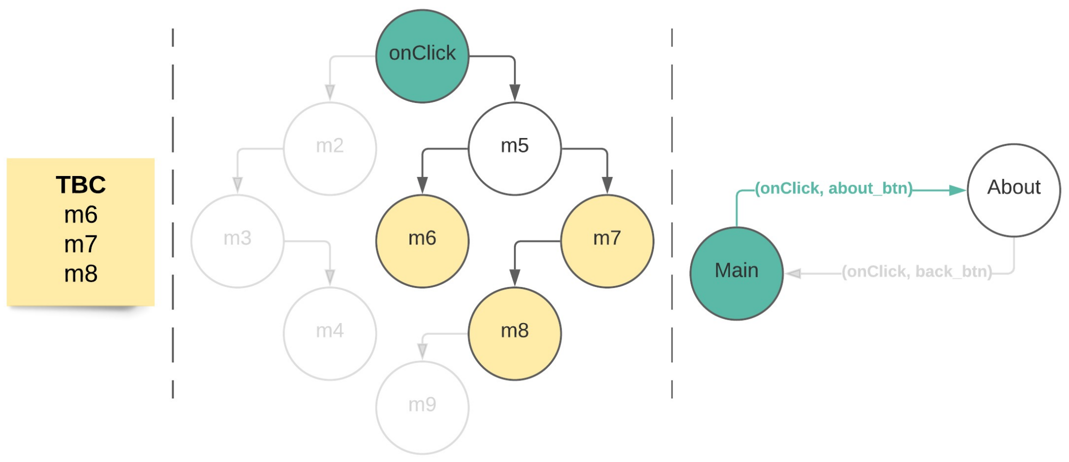


Figure 8 – Generation of the pruned navigational graph. From the left to the right we have a TBC, a pruned call graph, and a pruned navigational graph

connections to other nodes. It allows the trace of multiple alternative paths that achieve a common final destination. Each path starts from a node that is an entry-point of the APK and it ends in the screen navigation that covers the code change registered in the TBC. That is, only the last screen navigation covers the TBC. Suppose we have the following pruned navigational graph: $A \rightarrow B \rightarrow C \rightarrow D$. From that, suppose the transition $A \rightarrow B$ and the transition $C \rightarrow D$ covers the TBC. Then, ArcWizard will generate the following test scenarios:

1. $A \rightarrow B \rightarrow C \rightarrow D$;
2. $A \rightarrow B$.

Note that the first test scenario already exercises the transition $A \rightarrow B$. However, this was only a consequence of the route traced by ArcWizard to reach the last transition $C \rightarrow D$, which is the goal of this test scenario. The second test scenario was generated to ensure the coverage of the transition $A \rightarrow B$. So, there will be a set of test scenarios that ensure, as much as possible, the TBC coverage, regardless of whether there are test scenarios that cover more TBC due its route to reach its goal.

Figure 9 shows a pruned navigational graph where we can extract three test scenarios. Two test scenarios aim at navigating to the *Edit* screen with two alternative paths. The third test scenario is the navigation to the *About* screen. The edges in green highlight the moment the TBC will be covered. It is the act of navigating from the *Main* screen (the start point in blue) to the *Edit* or *About* screens (the final target screens) that exercises the method in the TBC. Note that we have the following navigation sequence: $Main \xrightarrow{onClick,list} List \xrightarrow{onClick,item}$

Details $\xrightarrow{onClick,edit\_btn}$ *Edit*. In this test scenario, only the last screen navigation, from *Details* to *Edit*, will exercise a method in the TBC. As we can observe, a complete path is a test scenario, and multiple test scenarios may have the same final target, covering the same code change.

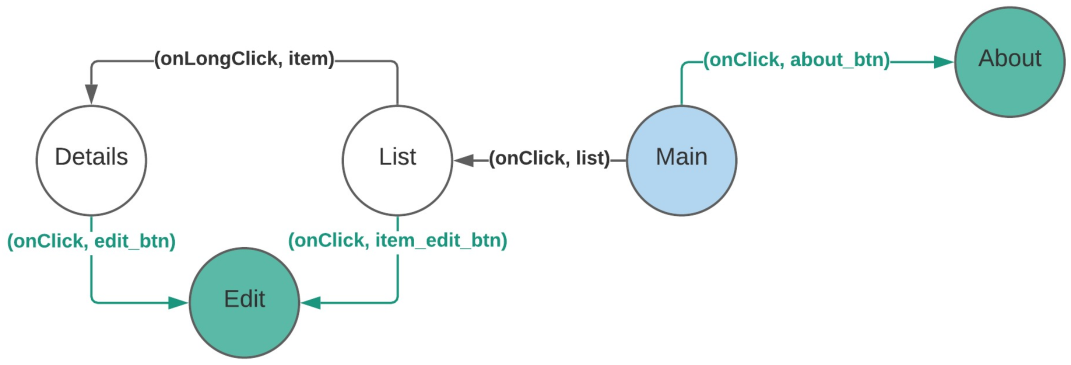


Figure 9 – Tracing a test scenario.

Test scenarios indicate a sequence of screen transitions (steps) that a tester can follow to reach the final screen navigation, the one that covers the detected changes. A test scenario step is composed of two screens (source and target), a GUI element (widget) that belongs to the source screen, and the GUI input event (action) that must be performed on the widget to navigate to the target screen. This action triggers the chain of methods where one or more of them are listed in the TBC. A single action of the final step has the potential of covering multiple methods. The widget is composed of a type and other attributes such as `android:text` and `android:id`. It is worth noting that the steps work as a chain, where the target of the previous step is the source of the next one, and so on. Figure 10 presents the excerpt of a single test scenario in JSON format. The test scenario holds a sequence of `steps`. A `step` has the `source`, `target`, `action` and `widget` attributes. The `source` and `target` are Android activities. The `action` is the callback method that will react to a user gesture. The `widget` is the GUI element that belongs to the `source` screen. It is the target of the `action` and it has its own attributes to identification on GUI. The `target` Android activity is the screen that will be opened when the `action` happens on the `widget` in the `source` Android activity. In this particular example, the first step indicates that a click event must happen on the *About* button in the *MainActivity*, which will drive the device to the *AboutActivity*. The second step indicates that a click event must happen on the *Back* button in the *AboutActivity*, which will drive the device back to the *MainActivity*.

We mentioned before that ArcWizard generates English-based test scenarios for exploratory testers. Such test scenarios are the final outcome of ArcWizard's process. The test scenario we present in Figure 10 is an intermediary version that matters for our purpose. This is our opportunity to create AETing.

```
{
    "steps":[
        {
            "source":"sample.app/sample.app.MainActivity",
            "target":"sample.app/sample.app.AboutActivity",
            "action":"onClick",
            "widget":{
                "type":"android.widget.Button",
                "attributes":{
                    "android:text":"About"
                }
            }
        },
        {
            "source":"sample.app/sample.app.AboutActivity",
            "target":"sample.app/sample.app.MainActivity",
            "action":"onClick",
            "widget":{
                "type":"android.widget.Button",
                "attributes":{
                    "android:text":"Back"
                }
            }
        }
    ]
}
```

Figure 10 – Test scenario sample.

Due to the static nature of a test scenario, the steps may indicate navigation that may not be possible for several reasons. That is, ArcWizard test scenarios are based on what is found during the code static analysis phase, regardless of the system runtime state to make the navigation possible. The steps are completely stateless. For example, a screen may require an internet connection, a sim card that has to be inserted into the device, or permission that must be enabled or disabled. None of these is known by ArcWizard when suggesting the test scenarios.

2.2.5 Coverage Detection

Beyond generating the TBC, and the Test Scenarios to assist exploratory testers in covering the TBC, ArcWizard also provides a built-in mechanism to identify in real-time the TBC coverage while the exploratory tester follows the test scenarios' suggestions. From the two given APK versions, an old and new version, this mechanism decompiles and instruments the new APK version annotating each method call with a special markup that ArcWizard uses to identify when a method from the TBC is executed. To avoid annotating the entire APK, which may cause failures in the APK after installation, ArcWizard only annotates the methods registered in the TBC. To enable this feature, the exploratory tester must keep the device connected to the host where ArcWizard is running and the instrumented APK must be installed in the device. Hence, ArcWizard can track the device's log and seeks the annotations made in the APK, and dynamically updates the TBC coverage information directly on the ArcWizard GUI.

This special mechanism is used during AETing's evaluations described in Chapter 4.

2.3 PAGE BROWSER

PageObject (FOWLER, 2013) is a design pattern that describes how an object-oriented programming language can effectively hide details of GUI screens implementation through classes APIs. It allows manipulations and interactions with GUI screens and their widgets without knowing their internal structure. For example, consider an APK GUI with a single screen that has a button and a text field. It would be modeled as a class with three methods: (1) a method to press the button; (2) a method to retrieve the content from the text field and (3) another method to set the content in the text field. Through this hypothetical class, the button in the GUI can be pressed by calling the corresponding method or the content of a text field can be retrieved through an accessor method as well. Figure 13 illustrates the example we just described.

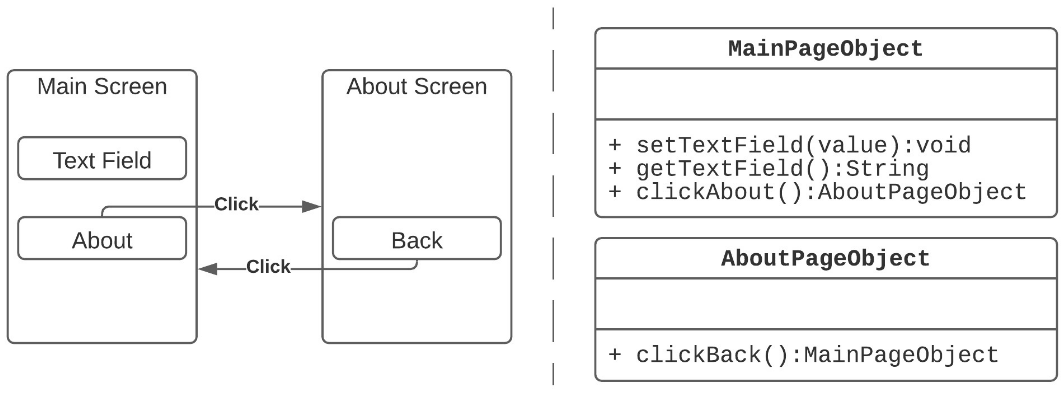


Figure 11 – Mapping of a GUI screen in a page object class.

A page object is able to do and see anything a human user can do and see on a software.

For instance, a page object can see a button in a given screen and click on it as a human user can. It encapsulates the mechanics required to find and manipulate GUI elements, which also involves the navigation between screens. Now suppose that the hypothetical APK has two screens and when pressing the button on the first screen it navigates to the second screen. The first page object related to the first screen would return the corresponding page object of the second screen.

Page objects are frequently used in testing. Test cases that use them have the benefits of avoiding to be updated when the GUI of the SUT changes. They bring a layer of interaction with the application GUI, which increases test code maintainability, reuse and reduces code duplication and coupling between test cases and applications (STOCCO et al., 2016). Testers that develop automated test suites based on a testing framework, which follows the PageObject pattern, have the benefit of not attaching the GUI interaction logic in the test code. It also improves the testers' productivity, since they can focus on the test implementation itself and not be concerned about GUI manipulation.

2.3.1 Framework overview

Page Browser is a Python-based testing framework from Motorola that follows the PageObject design pattern. A page object of Page Browser is a Python class implemented by programmers to map the APKs screens, their GUI elements (buttons, cards, text field, and others), and the navigation from/to other screens. Page objects also have metadata that register other information regarding the screen that is being mapped, such as: the APK version; the Android version; the carrier; the device model, and the global region. These metadata make it possible for the programmers to follow changes the APKs receive and reflect them in the page objects, at the same time they keep the compatibility across different versions. For instance, the Motorola APK called Moto Help enables fewer options on the main screen when it is installed in a low-end device and enables more options in high-end devices. It can be expressed as two different page objects for the same main screen but for two different devices.

Programmers implemented underlying mechanisms into Page Browser to deal with pop-ups, notifications, permission requests, and other visual elements that may break the automated test cases developed using Page Browser. This means that Page Browser carries what the testers need to automate a test case and avoid what they do not need or want to avoid when automating test cases.

Suppose that a *Main* screen has a button that drives a user to the *About* screen when it is pressed. And that the *About* screen has another button that drives the user back to the previous screen (Figure 12). It is a simple example where there are two screens interacting with each other. Now, Figure 13 shows a snippet of a Python code that implements the *Main* and *About* screens as page objects. We explain below this snippet.

The `@page_object` decorator marks a Python class as a page object and it receives a name. This name is the reference to use the page object instead of the class name. We have two page objects: `sample.app.main` and `sample.app.about`, both to the *Main* and *About* screens respectively. There is a second parameter related to the APK version. In this scenario, both page objects are mapping the screens for the first version of a hypothetical APK. The metadata allows Page Browser to instantiate in memory, during test execution, only compatible page objects based on the set of APKs installed in the test device. Other metadata can be registered as well. They help in better restricting the boundaries of the page object. It is possible to say that the page object is supported only by Android 9, for instance.

Each class in that snippet (`MainScreen` and `AboutScreen`) has the attribute `window` where the full activity package (the installed APK package concatenated with the activity package path) is referenced. The window attribute tells which APK screen the class is mapping. The GUI components from the screens are also implemented (`about_btn` and `back_btn`). Similar to the metadata in the `page_object` decorator, when mapping GUI widget components it is also possible to use any Android widget attribute (id, content-desc, text, and so on).

Finally, the screen navigation knowledge is registered using the `@go_to` decorator, which receives the page object's name destination. The `sample.app.main` page object has a route to navigate to the `sample.app.about` page object through the `about` method. The path back from the *About* screen to the *Main* screen is also defined through the `main` method, which belongs to the `sample.app.about` page object. It is worth highlighting that this navigation can be represented as a graph where nodes would be screens, the arrows would be the methods that are called when the navigation goes from the source screen to the target screen. This navigation knowledge supports AETing's tracing routing feature, which we describe later.

Usually, an APK has a screen that is defined as main, the entry-point of the APK. This is the screen that is opened when the APK is launched. A page object that maps an entry-point screen provides the API open that launches the APK. It is specially marked with the decorator `entry_point` as well (`MainScreen` class in Figure 13 is notably an entry-point). For instance, to launch the page object of *Main*, a tester has to make the following call:

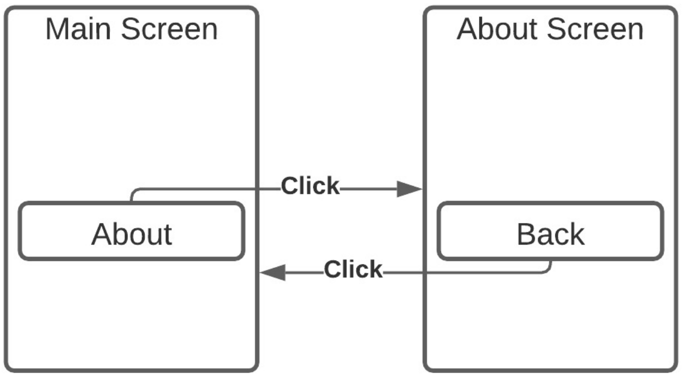


Figure 12 – Example of two APK screens where there is a relation in terms of navigation between them.

```python
@page_object('sample.app.main', version='1.0')
class MainScreen:
    window = 'sample.app/sample.app.MainActivity'
    about_btn = dict(text='About')

    @entry_point
    def open(self):
        launch_app()

    @go_to('sample.app.about')
    def about(self):
        self.about_btn.click()

@page_object('sample.app.about', version='1.0')
class AboutScreen:
    window = 'sample.app/sample.app.AboutActivity'
    back_btn = dict(text='Back')

    @go_to('sample.app.main')
    def main(self):
        self.back_btn.click()
```

Figure 13 – Page objects examples that model the *Main* and *About* screens from Figure 12.

`sample.app.main.open()`. Such a screen launching is transparent to the tester because it depends on the implementation in Page Browser page object. Page Browser programmers would apply any manipulation needed to launch the desired APK. It may raise directly the screen by intent or it may use the launcher to search and open the APK. APKs that show a wizard to set up at first usage may be also handled by the implementation when calling the open API.

2.3.2 Available resources

There are around two thousand page objects already implemented in Page Browser. This set of page objects maps screens required by Motorola test cases, which involves screens from

Motorola APKs, third-party APKs, and the Android platform as well. At the moment Page Browser connects with a device, it retrieves information from and matches them with the page objects metadata we mentioned previously to instantiate only page objects supported by the device connected to the host. It means that, in runtime, Page Browser uses only a subset of the overall implemented page objects.

3 STRATEGY

Our proposal consists of automatically mapping a test scenario provided by ArcWizard into elements of the Page Browser as close as possible, generating an automatic test case script. Recall from Section 2.2.4 that test scenarios are generated from static code analyses whereas the Page Browser elements come from programmers implementations of specific navigations based on PageObject (Section 2.3.1). Therefore, we have an already available library of page objects, but it may not be filled with enough elements needed by the ArcWizard's test scenarios.

3.1 OVERVIEW

To solve the matching problem introduced above, we map a subset of the Page Browser library as a CSP_M specification, define dynamically an refinement assertion between *Source* and *Target* screens, whose refinement consists of finding the best route (the shortest one) that leaves the main screen of the APK with the last step source screen of the ArcWizard test scenario as the target. It is worth noting, however, that the CSP_M specification is dynamically created when the device is connected. This is because only when the device is connected, the Page Browser can determine which elements of its library are available to that device (based on the metadata presented in Section 2.3.1). To improve the exploratory testing characteristic of AETing, we also add the Monkey testing tool (Monkey Documentation, 2019) to trigger pseudo-random Android system and user events at the last step source screen, expanding the interaction on the last step source of the ArcWizard beyond the last step action suggested, aiming to maximize coverage of code changes evolution.

Figure 14 overviews the AETing's workflow. ArcWizard receives two APK versions, identifies the differences between them, stores these differences in the set *To Be Covered* (Section 2.2.1) and generates test scenarios (to be exercised by testers). A test scenario is a set of screen navigation steps specifically designed to cover as much as possible the source code evolution between the provided APKs. The last step of a test scenario is responsible for such a coverage (Section 2.2.4). It has an action (an Android event such as `onClick`), a widget (a GUI element) that is the target of the event indicated, and a source screen (the APK GUI where the action must be executed) (Section 2.2.4). To start generating the automated test case (Section 3.2) that implements an ArcWizard test scenario, AETing loads all the Page

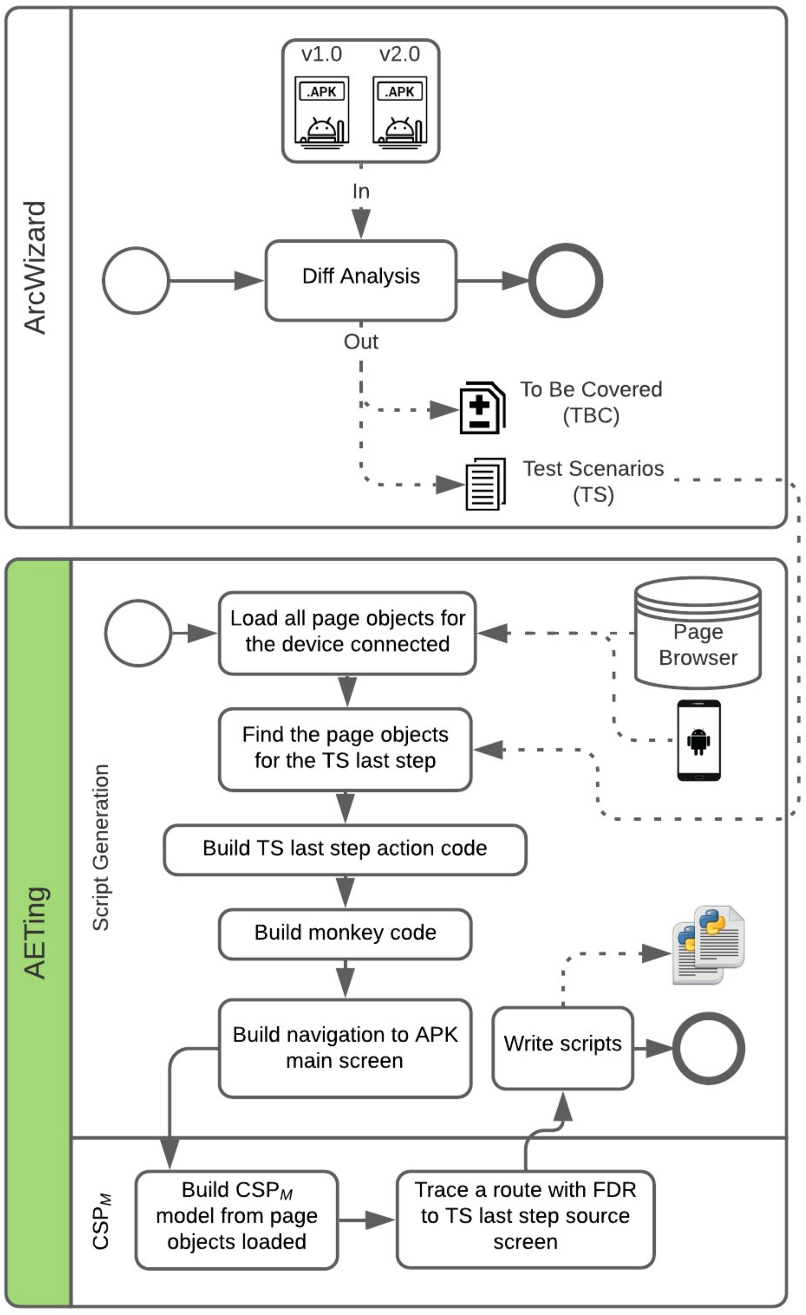

ArcWizard
v1.0
v2.0
.APK
.APK
In
Diff Analysis
Out
To Be Covered (TBC)
Test Scenarios (TS)
AETing
Script Generation
Load all page objects for the device connected
Page Browser
Find the page objects for the TS last step
Build TS last step action code
Build monkey code
Build navigation to APK main screen
Write scripts
CSP_M
Build CSP_M model from page objects loaded
Trace a route with FDR to TS last step source screen

Figure 14 – Workflow overview.

Browser page objects for the device connected to the host (Section 2.3.1). These page objects encode as Python classes the APK screens and their widgets. The last step of a test scenario is used to generate the code statement that performs the action over the widget indicated automatically. Since the attributes of the test scenario step are also mapped by the page object, AETing is able to find the page object that can perform the step. The Monkey is also added to improve the exploratory testing characteristic of AETing. To handle the screen navigations, AETing finds the page object that maps the main activity of the APK and uses it to navigate into the main screen. The final stage is to build the code that navigates from the main screen of the APK to the last step source screen, the screen where the last step action and Monkey will be executed. This stage consists in modeling the page objects loaded into CSP$_M$. This makes it possible to trace routes for a given pair of screens (source and target), and call FDR to find a route, from the available resources. In our case, the source is the main screen of the APK and the target is the source screen of the last step of the test scenario. Lastly, AETing generates the script files that cover the code evolution using each piece of code statement generated.

Our automated test cases have two missions: (1) perform the action suggested by the last step and (2) explore the source screen of the last step. This way, we firstly want to ensure the TBC coverage, then we explore the screen. For both missions to succeed completely, the device must have the source screen of the last step correctly reached and opened. To properly cover this precondition, two conditions must be satisfied: (1) Page Browser API `.open()` must drive the device to the main activity of the APK; (2) FDR must find a counterexample that traces a route, from the main activity to the source screen of the last step, and Page Browser must follow this route. Once the device navigates to the expected screen, our automated test cases have the context needed to perform their missions. The first mission is achieved by executing the last step action of the ArcWizard test scenario via the Page Browser, which can fail. The second mission is achieved by running the Monkey testing tool in an attempt to improve the exploratory testing characteristic of AETing. If the Page Browser fails to drive the device to the last step source screen, the Monkey testing tool is simply run on a different screen. But this does not break our generated automated test case. The only drawback is that we can have less coverage. Due to this, AETing would work as a kind of car, taking the passenger Monkey as close as possible to the destination, where Monkey will get off the car and perform its pseudo-random user and system events. Figure 15 presents the behavior of the automated test case.

In what follows, we describe AETing in more details.

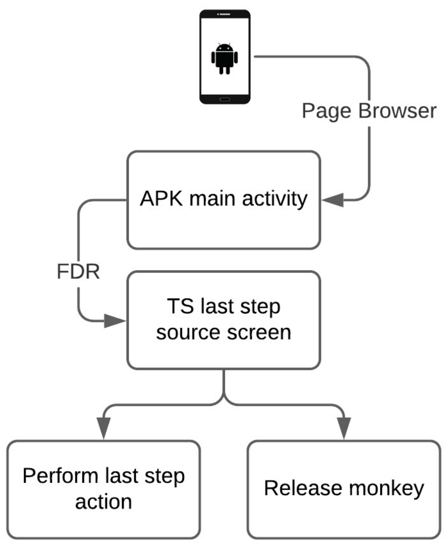


Figure 15 – Automated Test Case Structure.

3.2 AUTOMATED TEST CASE INCEPTION

When AETing receives the test scenarios from ArcWizard, it starts composing the initial structure of the automated test cases. This structure owns the code statement that performs the last step action, the code statement that triggers the Monkey, and the code statement that navigates to the main activity of the APK under execution. Below we describe how these code statements are generated.

Firstly, AETing loads all page objects from the Page Browser for the connected device, which usually takes less than 5 seconds. Then, AETing gets the last step source screen of a test scenario and finds among the loaded page objects the ones that implement that source screen. This source screen may be an Android Fragment, or even a WebView, for example. These elements can hold multiple screens into a single view (they give the usage experience of browsing over different screens, but it is a single one), unlike a regular Android Activity, which represents a real single view. Thus, Page Browser may return one or more page objects for a single screen from the test scenario. This means a test scenario may result in multiple automated test cases. If no page object can be found, it means the step is not supported. Figure 16 illustrates AETing's initial phase. In blue we have the page objects supported by the connected device. In yellow we have the page object that implements the source screen of a test scenario. For simplicity, the test scenario steps are represented as the source screen.

Secondly, for each page object found, AETing initializes one automated test case structure.

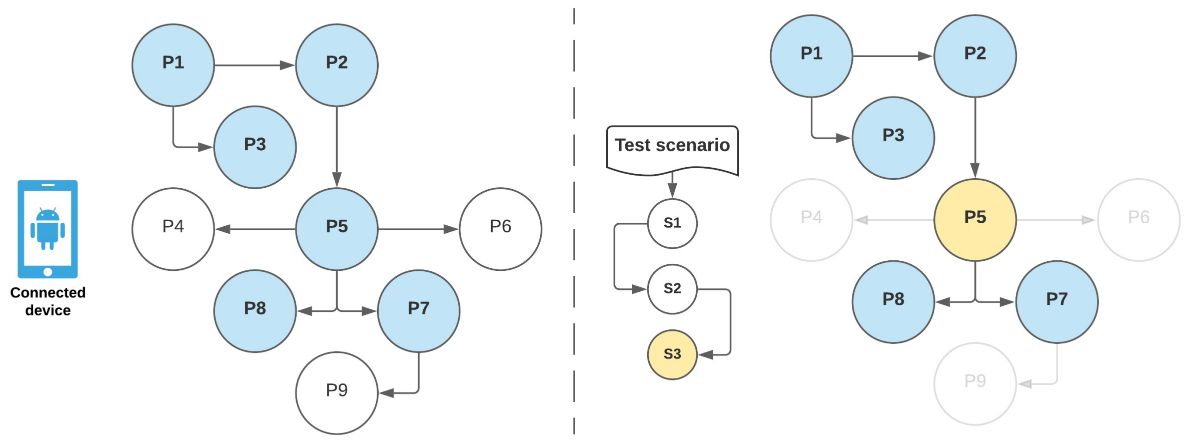
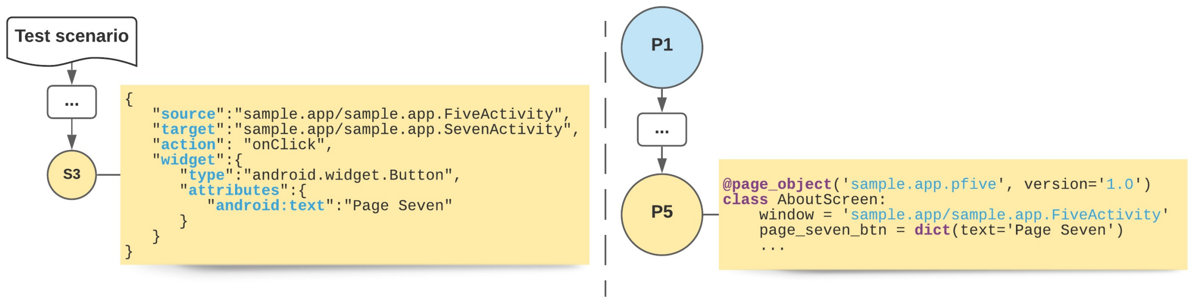


Figure 16 – AETing loads the page objects supported by the connected device (in blue) and finds among them the page object that maps the last step source screen of the ArcWizard test scenario (in yellow).

The widgets mapped in the page object of an automated test case are used to match the last step widget. The matching is based on the values of the attributes of the last step widget with the values of the attributes implemented on the widget of the page object. Considering the test scenario presented in Figure 10 and the page object `AboutScreen` presented in Figure 13, suppose we try to generate the code to perform a click on the widget Back. The matching result will be the following call `sample.app.about.back_btn.click()`. Now suppose the widget Back is not mapped in the page object. In this case, AETing becomes responsible for converting the last step widget attributes as parameters for Page Browser, resulting in the following call `sample.app.about.click(text="Back")`. Note that, for the first case, the code statement generated is the result of the matching of the widget as implemented by Page Browser programmers. For the second case, the code statement generated is the attempt of AETing in producing an equivalent, and fully supported, code statement to the Page Browser programmers widget mapping. Once this code is generated, it is appended to the automated test case structure. Following, the Monkey trigger command is built and appended to the automated test case structure as well.

Figure 17 illustrates a match between the ArcWizard test scenario last step and its corresponding page object. The match happens at two levels:

1. Find the page object based on the `source` attribute of the last step with the `window` attribute of among the page objects available for the connected device;
2. After finding the page object, the `widget` attribute of the last step is used to find the attribute in the page object that implements it.



```
{
   "source":"sample.app/sample.app.FiveActivity",
   "target":"sample.app/sample.app.SevenActivity",
   "action": "onClick",
   "widget":{
      "type":"android.widget.Button",
      "attributes":{
         "android:text":"Page Seven"
      }
   }
}
```

```
@page_object('sample.app.pfive', version='1.0')
class AboutScreen:
    window = 'sample.app/sample.app.FiveActivity'
    page_seven_btn = dict(text='Page Seven')
    ...
```

Figure 17 – AETing matches the widget of the last step of the test scenario with the implementation of this widget in the page object.

Finally, the first code statement navigation is built to drive the device to the main activity of the APK under execution. To do this, AETing retrieves from Page Browser the page object that maps the main activity of the APK and calls the API `.open()`. Still considering the test scenario presented in Figure 10 and the page object `MainScreen`, instead of the `AboutScreen`, presented in Figure 13, the matching result will be the following call `sample.app.main.open()`.

From this point, we have automated test cases that are able to open the APK, to perform the last step action over the target widget, and also to release the Monkey. Figure 18 presents the first version of a hypothetical automated test case generated by AETing, but it still lacks the behavior of driving the device from the main activity to the source screen of the last step. The building of this bridge to the last step source screen is done by FDR, described in the next sections.

The automated test case missions are to perform the ArcWizard last step action and release the Monkey, both at the correct screen. After executing one of the missions, we expect the device will no longer be in the correct screen. However, there is no guarantee of such success since the ArcWizard suggestion is based on static analysis (Section 2.2.4), which means AETing does not have information about the system state that would make the suggestion succeed. Thus, the script repeats the navigation before each mission as the attempt to ensure the device will be in the correct screen to perform the mission.

```
import page_browser

class AETingTestCase:

    def pre_setup(self):
        self.device = page_browser.connect()

    def run(self):
        self.perform_arcwizard_last_step_action()
        self.perform_monkey()

    def navigation_to_main_screen(self):
        self.device.sample.pone.open()

    def navigation_to_arcwizard_last_step_source_screen(self):
        # TODO: TO BE IMPLEMENTED DURING TRACING ROUTES
        pass

    def perform_arcwizard_last_step_action(self):
        self.navigation_to_main_screen()
        self.navigation_to_source_screen()
        self.device.sample.app.pfive.page_seven_btn.click()

    def perform_monkey(self):
        self.navigation_to_main_screen()
        self.navigation_to_source_screen()
        self.device.adb('monkey -p sample.app')
```

Figure 18 – The first version of the automated test case generated by AETing that is able to navigate to the main activity, perform the action suggested by ArcWizard and also release Monkey.

3.3 SCREEN NAVIGATION MODEL: A CSP$_M$ SPECIFICATION

In this section, we introduce CSP (SCHNEIDER, 1999) and FDR (GIBSON-ROBINSON et al., 2014), these are the technologies used by AETing to create the screen navigation model and to trace a route on the navigation model built.

3.3.1 CSP

Communicating Sequential Processes (CSP) is a process algebra based on events, processes, and operators over those events and processes. It allows modeling and analyzing concurrent and reactive systems (SCHNEIDER, 1999) and through the same operators, it is also possible to model a multi-agent system where its components interact with each other (ROSCOE, 1995).

We can start to describe elements of interest in CSP by defining processes and labeled transitions. For instance, to model a light switch we could define the processes *ON* and *OFF*, and the transitions could be *up* and *down*. $OFF \xrightarrow{up} ON$ and $ON \xrightarrow{down} OFF$ can be defined as the possible transitions of the light switch model. It is equivalent to say that a transition is a system state change. The process transitions would continue with subsequent transitions according to the model defined. In our simple model, we have a loop that is continuously executed: *OFF* $\xrightarrow{up}$ *ON* $\xrightarrow{down}$ *OFF*... . Figure 19 illustrate the model we just described.

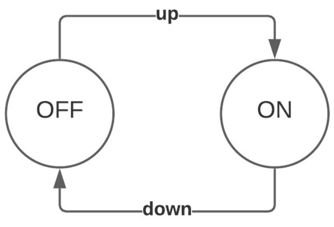


Figure 19 – Light switch model in CSP.

For the transition to happen, an *Event* must occur, which is represented by the labeled arrow. The transition *OFF* $\xrightarrow{up}$ *ON* means that the execution of *OFF* starts with the occurrence of the event *up* and the subsequent behavior is process *ON*. The event *up* is an interface of process *ON*. It means the event *up* performs a transition and then behaves as process *ON*.

Model checking is obtained through the tool FDR (Failures Divergences Refinement) (GIBSON-ROBINSON et al., 2014). FDR was designed to support a version of the dialect CSP called machine-readable CSP (or simply CSP$_M$). To verify a refinement, a semantic model must be chosen. There are three of them: *traces*, *failures* and *failures-divergences* (ROSCOE, 2005). In this work we just need to use the traces semantic model, which is computationally simpler and it is directly related to testing (NOGUEIRA et al., 2016) (CARVALHO et al., 2015).

CSP$_M$ is composed of a functional language to deal with data structures and a behavioral language based on the CSP events and operators. Through it we can: declare structured or enumerated data types using the keyword `datatype`; create events, which are the building-blocks over observable actions of a system, through the keyword `channel`; define processes to describe a system behavior. Besides processes that can use `datatype` and `channel` to describe behaviors, there are other definitions (operators), as such: *Prefix*, *External Choice*, *Generalized Parallel*, *Rename*, *Traces Refinement*, *Mutual Recursion*. These are just a few operators and constructors supported by CSP$_M$, but the complete list is presented in (ROSCOE, 2005) and (HOARE, 1985). We concentrate on describing those operators and constructors related to our work only.

3.3.1.1 Data type

A Data type defines named constant values or structured data through the keyword `datatype`. Following a simple statement where we define three values of programming languages.

```
datatype Languages = Java | Python | Go
```

Figure 20 – Defining data type.

3.3.1.2 Channel

Channels are the basis for defining events. They can be simple or complex. A simple channel definition is simply an event. A complex channel defines a base to instantiate an event described by `c.v`, where c is the channel name and v the message value to be communicated. It becomes an event when the values required are given. The values used in channels can come from data types as well. Figure 21 exemplifies the usage of the constructor `channel` to define simple and complex channels.

```
datatype Backend = Java | Python | Go
datatype Frontend = JavaScript | HTML | CSS
channel up, down
channel language: Backend
channel stack: Backend.Frontend
```

Figure 21 – Defining channels.

To use the channels up or down we simply specify their names. To use the channel `language` we can write as `language.Java` or `language.Go`. To use the channel `stack` we write `language.Java.HTML`. These usage examples are values of type Event.

3.3.1.3 Processes

Processes are the basic units to describe behaviors in CSP. They are self-contained entities that interact with each other through events (atomic actions performed or suffered by the processes). Describing systems in CSP implies that we compose a system with those processes where the system itself becomes bigger and still a self-contained entity.

CSP provides not only the infrastructure to define processes but it already has built-in processes that can be used as well. An example of a built-in process is `STOP`. It is a deadlock or broken behavior process that no longer communicates. Another built-in process is `SKIP`, used to terminate with success an execution. There is a built-in operator used when defining a process called *Prefix* (we present more operators in subsequent sections). It communicates an event and then it behaves as the process. The prefix syntax is `e -> P`, where `e` is an event, `P` is the process and the `->` is the prefix operator that binds both event and process. Figure 22 describes a coffee machine. Firstly, it communicates the event `coin`, then it behaves as the process `coffee -> SKIP`. This last process communicates the event `coffee` and reaches its final state `SKIP`, ending successfully.

```
channel coin, coffee
CoffeeMachine = coin -> coffee -> SKIP
```

Figure 22 – Defining a CSP process.

3.3.1.4 Parameters

CSP processes can receive a fixed number of parameters, which can be of any built-in type (*Numbers*, *Chars*, *Events*, *Tuples*, etc.) or data type previously defined. Figure 23 shows an example of a process `P` that receives a single numeric parameter `s`. It increments in one the input number given and calculates the remainder of the integer division by five. Thus, although the process `P(s)` can allow any integer to be given as input, the recursive call only considers calls between 0 and 4.

```
channel a
P(s) = a -> P((s + 1) % 5)
```

Figure 23 – Defining a CSP process that receives a *Number* type as parameter.

3.3.1.5 Recursion

CSP supports recursion through the usage of the prefix operator on the right side of the equation. It allows the description of an entire behavioral process based on shorter processes notations. Figure 24 presents a simple clock that continuously acts and interacts with its environment doing tick.

```
channel tick
Clock = tick -> Clock
```

Figure 24 – Recursive definition of a CSP process.

The *Mutual recursion* is an extra case where processes are bound to others. Usually, processes that apply mutual recursion are defined as a family of processes. That is useful to have a single process entry-point for a sequence of executions that produces the desired behavior. Figure 25 illustrates a family of process P that is defined in terms of other processes P, in each case is a function with one parameter.

```
channel go_to
P(0) = go_to -> P(1)
P(1) = go_to -> P(2)
P(2) = go_to -> P(0)
```

Figure 25 – Mutual recursive definition of CSP processes.

3.3.1.6 *External Choice*

The External Choice operator ([]) allows the environment to control the choice between the events of the processes that participate in the choice operation. The process P [] Q allows the environment to perform a choice among the events initially offered by P and Q. The environment would select process P if its events are possible to perform. Otherwise, the environment selects Q. The choice operation will behave accordingly to the selected process. In the case the same event from both P and Q is performed, the choice becomes non-deterministic. Nevertheless, the choice will not happen for the case where none event is possible to be selected. Figure 26 defines a set of mutually recursive processes where process B is defined in terms of a non-deterministic external choice. It raises the possibility of behaving as go_to -> C or go_to -> A, decided by the process itself.

3.3.1.7 *Renaming*

The Renaming operator ([[... <- ...]]) allows the substitution of events of a given process based on others. The following renamed process P[[a <- b]] substitutes the event

```
channel go_to
A = go_to -> B
B = go_to -> C [] go_to -> A
C = go_to -> D
D = go_to -> A
```

Figure 26 – Usage of *External Choice* operator.

a that process P can perform for event b, obtaining a new process without actually defining one that would communicate the event b from scratch. Figure 27 presents process A, which communicates the sequence of events $\langle a, b, a \rangle$ before behaving as SKIP. Process B behaves similar to A, but all events a communicated will be replaced by events c instead.

```
channel a, b, c
A = a -> b -> a -> SKIP
B = A[[a <- c]]
```

Figure 27 – Usage of *Renaming* operator.

3.3.1.8 Generalized Parallel

Generalized Parallel (`[||]`) forces the interaction of the participating processes with each other to synchronise on a set of events. This interaction, also named as synchronization, happens through the events that belong to a given set of events. The construction of a generalized parallel is exemplified as `P [|E|] Q`, where E is a set of events that will be synchronized in the participating processes. Below we list an example where the synchronization *must* happen and another where the synchronization *must not* happen.

1. `(p -> q -> P) [|E|] (p -> q -> Q), {p, q} in E` : In this example, the events of both processes that surround the set of events E *must* synchronize, since the events p and q are in E.

2. `(p -> q -> P) [|E|] (p -> q -> Q), {p, q} not in E` : In this example, the events of both processes that surround the set of events E *must not* synchronize, since the events p and q are not in E.

In Figure 28 we describe processes A and B that communicate the same sequence of events (`a -> b`). The process C offers the synchronization alphabet, composed of the event a, for P and Q. Then process C will communicate a single occurrence of event a, but two occurrences of event b as long as only the event a is in the synchronisation set.

```
channel a, b
A = a -> b -> SKIP
B = a -> b -> SKIP
C = A [|{a}|] B
```

Figure 28 – Usage of *Generalized Parallel* operator.

3.3.1.9 *Traces Refinement*

As we mentioned before, a CSP process models behavior that can be semantically described through three models. In our case, the *traces* model describes a process P as a set of sequences of events obtained by *traces(P)*. Following we can see traces for simple processes:

- *traces(SKIP)* $= \{<>, <\checkmark>\}$;
- *traces(STOP)* $= \{<>\}$;
- *traces(a* $\rightarrow$ *b* $\rightarrow$ *STOP)* $= \{<>, <a>, <b>, <a,b>\}$.

To express that a process *Q* refines a process *P* in the traces model we use the refinement relation `P [T= Q`, which mathematically means *traces(Q)* $\subseteq$ *traces(P)*. That is, if process *Q* does at most the same trace-based behavior of process *P*, then *Q* refines *P*. Traces Refinement (e.g: `S [T= I`) checks if traces of process I, namely implementation, is a subset of the traces of process S, namely specification. Figures 29 illustrates an assertion call that checks if process B refines process A. In this particular example, the assert fails since A communicates the event c, which is not communicated by B.

```
channel a, b, c
A = a -> b -> c -> SKIP
B = a -> b -> SKIP
assert B [T= A
```

Figure 29 – Usage of *Traces Refinement*.

Finally, an interesting point about refinements and the model-checking technique, supported by the tool FDR, is when a refinement is not satisfied. In this case, a counterexample is generated to help the specifier to identify the reason the refinement did not hold. In Figure

34 we show the counterexample of Figure 29. The counterexample shows the trace until the moment of the divergence, in event c. We will use such counterexamples in our favor.

```
{
    [...]
    "event_map":{
        "0":"0",
        "3":"a",
        "4":"b",
        "5":"c"
    },
    [...]
}
```

Figure 30 – Counterexample generated from Figure 29.

3.3.2 Building the navigation model

As we said previously, we use FDR to try finding a route from a source to a target screen. The source screen is the main activity page object of an APK. The target screen is the source screen of a test scenario last step from ArcWizard. But to get the corresponding implementation of such a navigation, we use page objects. So, in this section, we show how we create a CSP_M specification which encodes our current page objects' library.

As we exemplified in Figure 13, page objects have two especial Python decorators: `@page_object`, which holds the page object name and `@go_to`, which holds the name of the page object. Based on this structure, we know where a page object can drive the device to. The page object `sample.app.main` can drive the device to `sample.app.about` through the `about()` method call. The page object `sample.app.about` can drive the device to `sample.about.main` through the `main()` method call. Based on this markup structure, AETing scans every page object, earlier loaded from Page Browser supported by the connected device, looking for the `@go_to` decorator to create the *Link Structure* of Figure 31.

```
{
    "sample.app.main-sample.app.about":"about",
    "sample.app.about-sample.app.main":"main"
}
```

Figure 31 – Link Structure created from Figure 13.

The Link Structure is a JSON data structure that maps all the page objects navigation pairs available for the connected device. For each pair of page objects it stores which methods from the source page object drives the device to the target page object. Figure 31 shows the Link Structure we created from Figure 13. Note that the dash character (-) is used as a separator for the pair of page object names. Since the page object name is used to use it instead of the class name (Section 2.3.1), there is no chance of the dash being part of the page object name, since it is not supported by Python syntax. The page object name before the dash character is the source page object, the page object name after it is the target. These two page object names compose the *key*. The *value* is the method name that belongs to the source page object that drives the device from the source to the target page object.

We can now derive a CSP$_M$ specification based on the Link Structure. From the elements illustrated in Figure 32 (a hypothetical representation of the concrete elements of Figure 31) we create the CSP$_M$ specification in Figure 33. This CSP$_M$ specification allows us to trace a route for a given pair of page object names through FDR refinement. Each snippet of the CSP$_M$ specification is briefly explained below.

```
{
    "P1-P2":"method_pone_ptwo",
    "P1-P3":"method_pone_pthree",
    "P2-P5":"method_ptwo_pfive",
    "P5-P7":"method_pfive_pseven",
    "P5-P8":"method_pfive_peight",
}
```

Figure 32 – Generic Link Structure sample. To simplify the sample, P1, P2 and the other Ps are replacing the page object names `sample.app.pone`, `sample.app.ptwo` and so on.

```
datatype Screen = P1 | P2 | P3 | P5 | P7 | P8

channel Found
channel trans: Screen.Screen

Path(P1) = [] dest: {P2, P3} @ trans.P1.dest -> Path(dest)
Path(P2) = trans.P2.P5 -> Path(P5)
Path(P3) = trans.P3.P1 -> Path(P1)
Path(P5) = [] dest: {P7, P8} @ trans.P5.dest -> Path(dest)
Path(NB) = STOP

ReachFromTo(start, end) =
    let toEnd = {trans.X.end | X <- Screen}
    within Path(start) [|toEnd|] ([] ev:toEnd @ ev -> Found -> STOP)

assert Path(P1) [T= ReachFromTo(P1, P5)
```

Figure 33 – CSP$_M$ specification that encodes the generic Link Structure in Figure 32

1. Definition of `datatype Screen` along with all values, which are the page object names, the nodes in the graph;
2. Definition of event `Found` to indicate when FDR finds a transition to the target screen;

3. Definition of a channel `trans: Screen.Screen`, which indicates the transition from a screen to another (source and target, respectively);

4. Definition of processes `Path`, which is a complete path from a source screen to a target screen. If more than one target exists, an indexed external choice operator (`[]`) is used to capture the possibilities in a set;

5. The process `Path(NB) = STOP` is used to create a total definition. That is, if a `Path` does not exist, `STOP` is returned as a leaf terminal;

6. Process `ReachFromTo` receives two screens `start` (source) and end (target) to search for the last transition that reaches the target. `toEnd` is a temporary set of events that allows the transition to target. The parallel operator (`[|toEnd|]`) monitors if a target transition to the target screen exists. If so, the event `Found` is offered to denote a special finding. This event `Found` is key to understand the refinement assertion described in what follows;

7. The assertion uses a traces refinement (`[T=`), the set of traces of the process `ReachFromTo(P1, P5)` may be a subset of the set of traces of the process `Path(P1)`. When a special case (which includes the event `Found`) happens, this refinement always fails, providing us a counterexample trace which is used in our mapping strategy.

In the next section we use the CSP$_M$ specification presented in Figure 33 to generate a counterexample and integrate such an outcome into the AETing's strategy.

3.3.3 Tracing routes to target screen

The step structure of the test scenario (Figure 10) holds the window paths for both source and target screens. This information is implemented in the page objects, as we presented in Figure 13, to allow the search for a page object from a test scenario step.

As mentioned earlier, the focus of AETing is using the last step of a test scenario, since this is the one that covers the code change from the TBC. In Section 3.2 we started building the automated test cases and in Section 3.3 we built the CSP$_M$ navigational model that allows us to trace a route between two page objects. Here we describe how AETing generates the code that navigates from the main activity of an APK to the last step source screen, the final destination where the last step action and the Monkey must be performed.

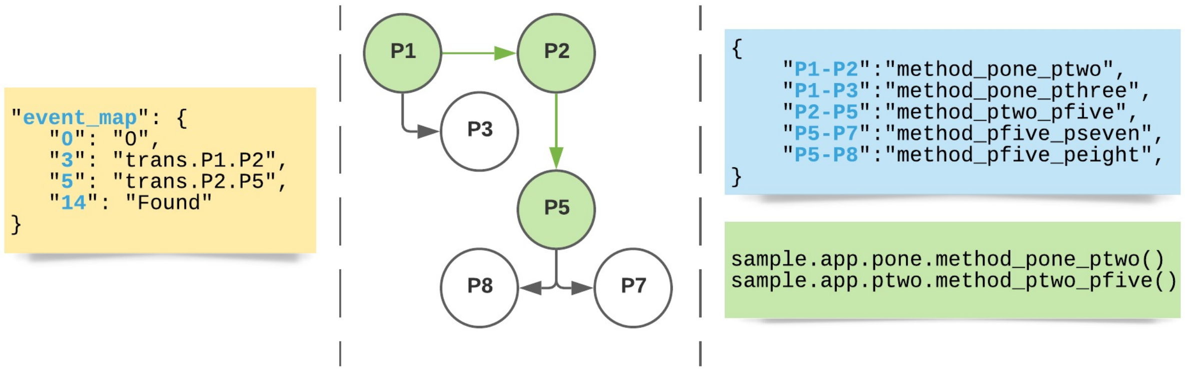
The automated test cases' initial structures already have the name of the page objects that refer to the main activity of the APK and the last step source screen. With the CSP$_M$ navigational model, we are able to ask FDR to trace a route between these two page objects. The routing works in two cascading stages of increasing difficulty: **Stage 1**, which does not require FDR usage and **Stage 2**, which does require FDR usage. This separation exists to avoid unnecessary FDR calls. Following we detail each stage:

- **Stage 1**: The first effort is checking whether there is a direct link from the main activity screen S1 to the last step source screen S2. If AETing finds an entry with S1-S2 in the Link Structure, this route is resolved easily;
- **Stage 2**: If Stage 1 does not succeed, AETing changes the refinement assertion of CSP$_M$ model with the main activity screen S1 and last step source screen S2 and calls FDR to find a possible route from S1 to S2. This can take some time but, in most cases, less than a second.

When FDR finds a route, it produces a trace (counterexample) of connected screens (complete navigation) to reach the final target, in our case from the main activity to the source screen of the last step of the ArcWizard test scenario. We exemplify executing the CSP$_M$ model illustrated in Figure 33, which traces a route from P1 to P5. It produces the counterexample presented in Figure 34. In yellow we have the FDR counterexample listing the events that occurred during tracing. Note the transitions appear in pairs of page objects. The nodes in the graph representing the path traced by FDR. In blue we have the *Generic Link Structure* to help in finding each method call for each pair of page object names. In green, we have the code that drives the device from P1 to P5. This is the missing piece that will complete the automated test case example of Figure 18.

When FDR does not find a route, AETing does not set on the automated test case structure the code statement that drives the device from main activity to the source screen of the last step of the ArcWizard test scenario. Possible reasons for FDR not finding a route are: (1) lack of intermediary page objects between P1 and P5; (2) miss-implementation of the go_to decorator.

Since one of the missions of our automated test cases is to perform the last step action on the last step source screen, it will not be able to perform such an action. The second mission of our automated test cases is to perform the Monkey testing on the last step source screen



```
"event_map": {
   "0": "0",
   "3": "trans.P1.P2",
   "5": "trans.P2.P5",
   "14": "Found"
}
```

```
{
    "P1-P2":"method_pone_ptwo",
    "P1-P3":"method_pone_pthree",
    "P2-P5":"method_ptwo_pfive",
    "P5-P7":"method_pfive_pseven",
    "P5-P8":"method_pfive_peight",
}
```

```
sample.app.pone.method_pone_ptwo()
sample.app.ptwo.method_ptwo_pfive()
```

Figure 34 – FDR counterexample of Figure 33 that traces a route from node P1 to node p5.

as well. However, in the case where it will not be possible to drive the device to the expected screen, the Monkey testing will be triggered on the main activity.

In any of the circumstances we just described, AETing will generate the automated test case script. However, the completeness of the script depends on the success of FDR in finding a route. If FDR finds a route, the script will be complete. If FDR does not find a route, the script will not be able to drive the device from the main activity to the source screen of the last step of the ArcWizard test scenario.

Figure 18 presented the first version of an automated test case that still lacks the navigation to the *FiveActivity*, which is implemented by the page object `sample.app.pfive`. In the *Generic Link Structure* (Figure 32) there is no direct path from P1 (*OneActivity*, the main actvity in this particular example) to P5 (*FiveActivity*, the ArcWizard last step source screen required by AETing automated test case). Thus, AETing calls FDR to trace a route from P1 to P5. Figure 35 shows the final version of an automated test case generated by AETing, which has now the implementation of the method `navigation_to_arcwizard_last_step_source_screen`.

```python
import page_browser

class AETingTestCase:

    def pre_setup(self):
        self.device = page_browser.connect()

    def run(self):
        self.perform_arcwizard_last_step_action()
        self.perform_monkey()

    def navigation_to_main_screen(self):
        self.device.sample.pone.open()

    def navigation_to_arcwizard_last_step_source_screen(self):
        self.device.sample.app.pone.method_pone_ptwo()
        self.device.sample.app.ptwo.method_ptwo_pfive()

    def perform_arcwizard_last_step_action(self):
        self.navigation_to_main_screen()
        self.navigation_to_arcwizard_last_step_source_screen()
        self.device.sample.app.pfive.page_seven_btn.click()

    def perform_monkey(self):
        self.navigation_to_main_screen()
        self.navigation_to_arcwizard_last_step_source_screen()
        self.device.adb('monkey -p sample.app')
```

Figure 35 – The final version of the automated test case generated by AETing that is able to navigate to the main activity, then to the last step source screen, perform the action suggested by ArcWizard and also release Monkey.

3.4 GENERATING THE AUTOMATED TEST CASES

The major phases of AETing to generate automated test cases that cover code changes are:

- **Phase #1**: load all page objects for the device connected;
- **Phase #2**: find the page objects for the test scenario last step source screen;
- **Phase #3**: build the test scenario last step action code statement;
- **Phase #4**: build Monkey code statement;
- **Phase #5**: build the code statement that drives the device to APK main activity;
- **Phase #6**: build the navigational screen model in CSP$_M$;
- **Phase #7**: trace a route that navigates from APK main activity to the last step source screen.

Recall from Section 3.1, the automated test case generated through this process has two missions:

- **First mission**: perform the test scenario last step action;
- **Second mission**: explore the test scenario last step source screen.

The first mission is the implementation of a test case based testing strategy, where we aim at a specific target. The second mission is the attempt to automate the exploratory testing strategy, where we trigger multiple events over multiple targets. These testing strategies have their own characteristics (SHAH et al., 2014) and the results produced by each have shown a slight differentiation (Itkonen; Mantyla; Lassenius, 2007) between them. AETing is a single solution that delivers both testing strategies to maximize the TBC coverage, introduced in Section 2.2.1. We also mentioned in Section 2.2.4 that multiple test scenarios may have the same final target, making it possible to generate duplicated automated test cases. For the human user of ArcWizard, each test scenario provides alternative paths to reach the target. Since they are a result of a static analysis, ArcWizard can not guarantee which path is possible to be executed. Thus, it provide the possibilities it could trace. For AETing, only the target matters. Thus, the duplicated scripts generated do not hold alternatives screens navigation, but they are equal indeed. Since execute duplicated automated test cases will increase the execution time and will not deliver any gain, AETing identifies such duplicated scripts and ensures the uniqueness of them by discarding the duplicates. Finally, automated test case files are generated.

4 AETINGS' EVALUATIONS

In this chapter, we analyze AETing, Monkey, and experienced human testers for the purpose of evaluation with respect to effectiveness (in terms of code coverage) and efficiency (in terms of time to complete exploratory sessions) in the context of an exploratory testing session exercising Motorola APKs.

It is important to mention that, due to the Motorola confidentiality policies, we are not allowed to share details regarding the resources and APK versions used during the experiments, but only the data strictly related to the empirical study itself [1].

4.1 EXPERIMENT: AETING VS MONKEY

In this section, we describe an experiment defined to compare AETing against Monkey in terms of TBC coverage exercising four Motorola APKs. For the purpose of organization, this section was divided into minor subsections as follows:

- Section 4.1.1 introduces the main concepts about empirical study and the statistical analysis techniques;
- Section 4.1.2 describes the research question, hypotheses and the variables of the experiment scope;
- Section 4.1.3 presents the experiment preparations and execution;
- Section 4.1.4 ends this experiment section with the data analysis and hypothesis testing.

4.1.1 Main concepts

In this section, we introduce the main concepts about the empirical study and the statistical analysis techniques. Such resources are the baseline of our experiment to compare AETing against Monkey.

[1] AETings' evaluations data: https://cutt.ly/rkp7yHI

4.1.1.1 Variables

When we are defining a formal experiment, it is part of the definition to identify and classify the elements from the environment where the experimentation will happen. These are the elements we want to measure and control to be able to achieve reliable results.

The variables in a formal experiment are the elements in the environment that we want to control and measure the outcomes. There are two kinds of variables: (1) *independent variable*, the one we want to control in an experiment, and (2) *dependent variable*, the one we want to measure the effect achieved when changing independent variables (WOHLIN et al., 2012).

As an example, suppose we want to measure the effect in the code coverage results applying different testing techniques. Here, the *code coverage* is our dependent variable and the *testing technique* is our independent variable.

4.1.1.2 Fixed independent variables and Factors

The experimentation environment contains elements that we must be aware of in order to control the effect in our dependent variable. As we mentioned before, those elements are called *independent variables*. Some independent variables are controlled at a *fixed level* during the experimentation. They are not the ones that we want to control in order to affect the dependent variables. Actually, we stabilize those independent variables to avoid undesired influence from the experimentation environment.

The independent variables are managed at two groups of importance: *fixed level*, the ones we do not want to manipulate, but we want to keep fixed during experimentation, and *factors*, the ones we do want to manipulate in order to affect the dependent variables (WOHLIN et al., 2012). Therefore, the factors are the independent variables that we intentionally vary while performing the experiment (JURISTO; MORENO, 2010).

Recall the experiment we started to describe in Section 4.1.1.1. We have the code coverage, as our dependent variable, and the testing technique as our factor. A third element to be added to this experiment definition is the fixed independent variable, which can be the test case to be executed.

4.1.1.3 *Treatments*

We introduced the factor notion previously in Section 4.1.1.2, which describes an independent variable that we want to control and observe the effect of such a control in the dependent variables.

As we can notice, factors are a specific kind of independent variable. However, as its definition describes, it is a kind of. It defines a class, so to speak, of elements from the experimentation environment. This way, factors are the classes or types, and the named *treatments* are the values of the factors (JURISTO; MORENO, 2010).

From the example in Section 4.1.1.2, we have our dependent variable (the code coverage), the fixed independent variable (the test case) and the factor (the testing technique). The treatments of the testing techniques can be *manual testing* and *automated testing* techniques.

4.1.1.4 *Objects*

We described until now the notion of the dependent variable, fixed independent variable, factors, and treatments. With this set, we know the elements we want to control and the elements we want to measure. In addition to the presented set, we define the notion of *object*. An object, also known as an *experimental unit* (JURISTO; MORENO, 2010), is the target that will be affected by the factors (WOHLIN et al., 2012).

From the example, we started in Section 4.1.1.1 and we incrementally added new elements to it, we can now add the object. The current state of the example describes an environment where we want to evaluate two types of testing techniques (factors) and measure the code coverage (dependent variable) each technique can achieve executing the same test case (fixed independent variable). The missing part here is the software to be tested, which is our object, the target of the factors.

4.1.1.5 *Statistical test*

During the execution of an experiment, we measure the dependent variable to have a collection of data that makes it possible to elaborate conclusions through analysis. Valid and clear conclusions are based on the results from the data statistically tested at a given level of significance (WOHLIN et al., 2012). A statistical test provides a tool that intends to confirm whether there is enough evidence to reject or accept hypotheses. The hypothesis testing can be two-tailed or one-tailed (HECKERT et al., 2012).

The two-tailed test verifies if a sample is greater than or less than a certain range of values from a distribution. We may also be interested in verifying only one of the extreme values at one side of the distribution using the one-tailed test. For this case, we have the one-tailed left test, which verifies if a sample is less than the extreme left range of values, and we have the one-tailed right test which verifies if a sample is greater than the right extreme range of values (Figure 36) (JURISTO; MORENO, 2010).

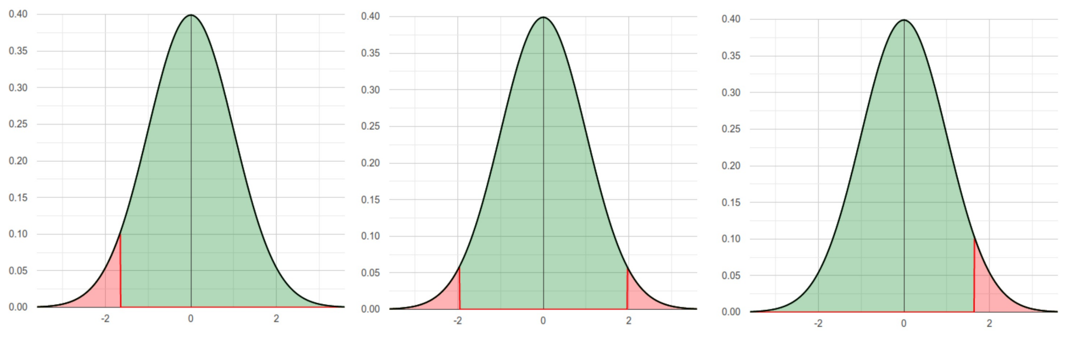


Figure 36 – From the left to the right, the one-tailed left test, two-tailed test, one-tailed right test.

In Figure 36 the red region is known as *alpha*. It determines the critical area of the distribution (significant level), which is the probability of rejecting a null hypothesis when it becomes true. The standard value of *alpha* is usually 5% (CRAMER; HOWITT, 2004). It stays in 5% for one-tailed tests, but it allots half of the *alpha* in each direction (left and right) for two-tailed tests. When *alpha* is 0.05, the test has a 5% chance of producing a significant result when the null hypothesis is correct. Thus, a correct hypothesis requires strong evidence to be rejected. As lower is the *alpha*, the greater is the exigency. If a sample falls into a critical area, the null hypothesis is rejected and the alternative hypothesis is accepted. The green region in Figure 36 points the confidence interval of 95% when the *alpha* is 0.05. It is the confidence interval that may stand a null hypothesis. If a sample falls into the confidence interval, the null hypothesis is not rejected and the alternative hypothesis is not accepted.

Hypothesis testing evaluates if it is possible to reject a given *null hypothesis* and to accept a given *alternative hypothesis* based on statistical testing. The null hypothesis describes a property from the collection of data resulting from the experiment execution that we want to say it is not true. The alternative hypothesis stands in favor of what it is being rejected by the null hypothesis. (WOHLIN et al., 2012).

The null hypothesis (H_0) and the alternative hypothesis (H_a) for our experiment example from Section 4.1.1.4 may be the following:

- (Null hypothesis) H_0: manual testing can achieve more code coverage than automated testing;
- (Alternative hypothesis) H_a: automated testing can achieve higher code coverage than manual testing.

We want to accept the alternative hypothesis (*automated testing coverage* $>$ *manual testing coverage*). To reject the null hypothesis and accept the alternative hypothesis, the experimentation must collect data to be statistically tested by a one-tailed right test in order to provide a safe platform for such a rejection and acceptance in the experiment defined.

The rejection or acceptance of the hypothesis relies on the statistical test and there are plenty of them. Nevertheless, for the purpose of this work, we will only present the *Mann-Whitney U test*, which is related to our experimentation.

4.1.1.6 Mann-Whitney U test

The statistical tests are divided into two major groups: parametric and non-parametric tests. The parametric tests assume that the sample of data given follows a normal distribution. Such a distribution produces a well-shaped bell when plotted. There are also tests that verifies the normality of a data set. The *Shapiro-Wilk test* is widely used for this purpose (DAS, 2016). On the other hand, the non-parametric tests make less, if any, assumptions regarding the sample data distribution. Thus, it is less powerful than the parametric tests (GRECH; CALLEJA, 2018).

To compare two samples of data we can apply *t-test*, assuming the data follows a normal distribution, that is, it is a parametric test; or we can apply *Mann-Whitney U test*, also known as *Wilcoxon Rank Sum test*. It is a non-parametric alternative to the *t-test* (WOHLIN et al., 2012).

Mann-Whitney U test allows the comparison of two samples of means. Thus, it was designed to compare two treatments of a given factor against each other. We can test if a result group of treatment A is less (one-tailed left), greater (one-tailed right), or different (two-tailed) from a result group of treatment B, for instance. As we mentioned before, this statistical test requires two groups of independent samples from the same population and these independent groups should not have a normal distribution (NACHAR, 2008). The outcome of the test is the *p-value*, which is the probability of determining if there is enough evidence to reject the null hypothesis based on the significant level.

For the One-sided left approach, a higher *p-value* supports the null hypothesis. That is, if we want to test if Group A is less than Group B, the null hypothesis will be $A > B$, the alternative hypothesis will be $A \leq B$. If *p-value* $>$ *alpha* the null hypothesis is not rejected. That is, we do not have enough data to say Group A is less than Group B.

For the One-sided right approach, a lower *p-value* supports the alternative hypothesis. That is, if we want to test if Group A is greater than Group B, the null hypothesis will be $A \leq B$, the alternative hypothesis will be $A > B$. If *p-value* $<$ *alpha* the null hypothesis is rejected. That is, we have enough data to say Group A is greater than Group B.

For the Two-sided approach, a lower *p-value* supports the alternative hypothesis. That is, if we want to test if Group A differs from Group B, the null hypothesis will be $A = B$, the alternative hypothesis will be $A \neq B$. If *p-value* $<$ *alpha* the null hypothesis is rejected. That is, we have enough data to say Group A differs from Group B.

Next, we define the experiment environment to compare AETing against Monkey based on the concepts introduced.

4.1.2 Planning

In this section, we describe the research question, hypothesis and the variables of the experiment scope that defines how the experiment will be executed in order to compare AETing against Monkey.

We draw an experiment that compares AETing, which is the execution of the ArcWizard suggestions and the release of Monkey at the target screen, against Monkey alone. This comparison is based on the execution of these exploratory methods covering the TBC. The research question is: *Can AETing cover more the TBC instead of Monkey*? Following our null and alternative hypotheses:

- (Null hypothesis) H_0: The Monkey average coverage of TBC is greater than the AETing average coverage (M_monkey $>$ M_aeting);
- (Alternative hypothesis) H_a: The AETing average coverage of TBC is greater than the Monkey average coverage (M_monkey $<$ M_aeting);

We analyze the coverage of the TBC for each *Exploratory method* when testing the Motorola APKs (FM Radio, Digital TV, Camera and Moto Help). This analysis is based on the effect of the execution in the dependent variable *Coverage of the TBC set*. For AETings' executions, we apply a different set of *ArcWizards' test scenarios* for each Motorola APK. For Monkeys' executions, we just execute it as is. Table 1 defines the variables. Table 2 presents the treatments of both factors, Exploratory method and ArcWizards' test scenarios.

Fixed independent variables	**Factors**	**Objects**	**Dependent variable**
Page Browser framework instance	Exploratory method	FM Radio	Coverage of the TBC set
Motorola Android device	ArcWizards' test scenarios	Digital TV	
		Camera	
		Moto Help	

Table 1 – Fixed independent variable, factors, and object considered for our experiment.

Treatments	
Exploratory method	**ArcWizards' test scenarios**
AETing	26 test scenarios for FM Radio
Monkey	25 test scenarios for Digital TV
	22 test scenarios for Camera
	102 test scenarios for Moto Help

Table 2 – Treatments for Exploratory method and ArcWizards' test scenarios factors. *The test scenarios are applicable only for AETings' executions.*

4.1.3 Operation

In this section, we describe the environment setup and execution of the experiment that compares AETing against Monkey.

A single test execution uses an Exploratory method and a Motorola Android device connected to a host where we have the Page Browser framework installed. The test execution exercises a Motorola APK and ArcWizard measures the Coverage of the TBC set during testing until the execution ends (ArcWizard's *Coverage Detection* feature introduced in Section 2.2.5). Figure 37 illustrates the experiment environment.

Each Motorola APK also has its own setup to allow testing most of its features automatically. We list below the setup for each Motorola APK:

- **FM Radio**: the device had a headset plugged, which is used as an antenna to capture radio signals;

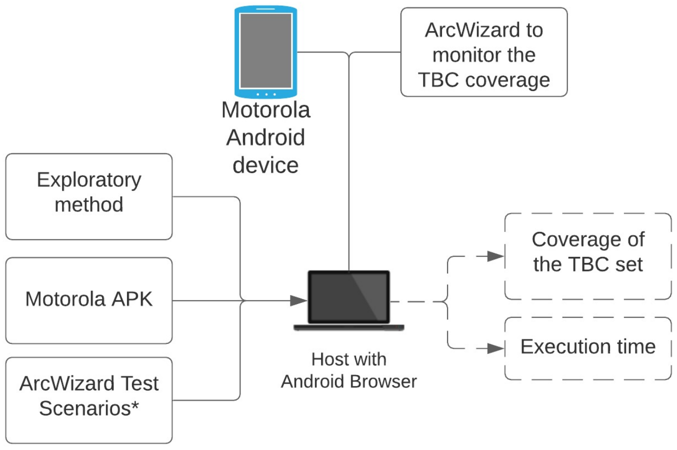


Figure 37 – Illustration of the experiment environment to compare AETing vs Monkey. ** The test scenarios are applicable only for AETings' executions.*

- **Digital TV**: the device had a headset plugged, which is used as an antenna to capture television signals;
- **Camera**: the device was pointing the back and front cameras to objects possible to enable the focus;
- **Moto Help**: the device was connected to a Wi-Fi network.

In order to collect data for statistical testing, we performed 20 executions for each Exploratory method for each Motorola APK. We executed AETing 20 times for FM Radio, another 20 executions for Digital TV and so on. The same is applied for Monkey. Before each particular execution, we cleaned up the APK cache and previous data to start a completely clean execution without interference from previous runs. It means our execution data are **independent samples**. It is an important attribute of our data for the choice of the statistical testing, as we will see in Section 4.1.4.

The execution duration was based on AETing's execution, since Monkey executes indefinitely until we stop it. Although the duration is not under evaluation, it is important to not have Monkey running longer than AETing for a proper comparison. For instance, if the first AETing's execution for FM Radio lasts 48 minutes, then the first Monkey's execution for FM Radio will also be restricted to 48 minutes only.

The number of methods of the TBC for each Motorola APK are: 143 methods for FM Radio, 477 methods for Digital TV, 271 methods for Camera, and 142 methods for Moto

Help. Table 3 presents the Coverage of the TBC set of the executions of AETing and Monkey testing the Motorola APKs.

	FM Radio		**Digital TV**		**Camera**		**Moto Help**	
	TBC: 143		TBC: 477		TBC: 271		TBC: 142	
	AETing	Monkey	AETing	Monkey	AETing	Monkey	AETing	Monkey
1	104	97	325	6	122	2	63	47
2	103	100	321	6	124	2	59	10
3	111	100	335	6	131	2	59	11
4	111	99	335	6	122	2	65	35
5	104	99	310	6	124	2	68	10
6	97	93	332	6	122	2	67	35
7	107	99	320	6	120	2	70	11
8	103	98	309	6	124	2	73	14
9	104	102	328	6	129	2	70	43
10	104	100	328	6	122	2	68	41
11	104	97	330	6	126	2	72	39
12	103	92	307	6	127	2	62	10
13	99	97	311	6	125	2	62	15
14	110	94	329	6	126	2	68	48
15	111	100	316	6	126	2	65	52
16	103	97	324	6	126	2	67	53
17	110	100	319	6	126	2	65	42
18	103	97	328	6	125	2	63	54
19	111	93	320	6	123	2	68	48
20	97	100	338	6	124	2	64	34

Table 3 – The Coverage of the TBC set for each Exploratory method execution for each Motorola APK.

4.1.4 Analysis

In this section, we analyze the data collected from the experiment execution (Table 3) and we apply hypothesis testing in order to evaluate our initial question: *Can AETing cover more the TBC instead of Monkey?*

To test our hypotheses, we need to apply a proper statistical test based on the nature of our data. We already know that the data collected are independent samples, since each execution started over the environment setup for each APK before a new execution (Section 4.1.3). We apply the Shapiro-Wilk test of normality to identify if our data set follows a normal distribution. Table 4 shows the normality test results for our data set.

	FM Radio		**Digital TV**		**Camera**		**Moto Help**	
	AETing	Monkey	AETing	Monkey	AETing	Monkey	AETing	Monkey
Normal distribution	No	No	Yes	No	Yes	No	Yes	No

Table 4 – Shapiro-Wilk test results for normality of the exploratory method coverage results

As we can observe, there is no pair of normal distribution data for each Motorola APK. Only the results of Digital TV and Camera have at least one set of data from an exploratory method (AETing in these two particular cases) that follows the normal distribution. Our goal is to compare two sets of data through statistical analysis, but our data do not follow the normal distribution, and they are independent samples. In this special case, we can apply the Mann-Whitney U test, which is a non-parametric test that does not assume the data given follows the normal distribution and it can compare two samples of data, as we described at Section 4.1.1.6.

Recall from Section 4.1.2, follows our hypotheses:

- (Null hypothesis) H_0: The Monkey average coverage of TBC is greater than the AETing average coverage (M_monkey $>$ M_aeting);
- (Alternative hypothesis) H_a: The AETing average coverage of TBC is greater than the Monkey average coverage (M_monkey $<$ M_aeting);

Let A_C be AETing's Coverage of the TBC set (Group A) mean, M_C be Monkey's Coverage of the TBC set (Group B) mean, *p-value* the outcome of the one-tailed right approach of Mann–Whitney U test over A_C and M_C, and $alpha$ be the significance level of 5% such that:

$$\begin{cases} Accept\ H_0, if\ \text{p-value} \geq alpha.\ It\ means\ A_C \leq M_C \\ Accept\ H_a, if\ \text{p-value} < alpha.\ It\ means\ A_C > M_C \end{cases}$$

The results in Table 5 indicate that AETing is the best Exploratory method in comparison with Monkey when covering the TBC for the Motorola APKs evaluated.

Despite the statistical testing results in Table 5 indicate AETing as the best for covering the TBC, we noticed in the raw results in Table 3 that Monkey achieved a Coverage of the

	FM Radio	**Digital TV**	**Camera**	**Moto Help**
p-value	3.94%	3.95%	3.71%	3.27%
H_0	Rejected	Rejected	Rejected	Rejected
H_a	Accepted	Accepted	Accepted	Accepted

Table 5 – For these four Motorola APKs, our alternative hypothesis was accepted, and the null hypothesis was rejected.

TBC set close to AETing for FM Radio and Moto Help, at same time it did not have success when testing Digital TV and Camera. Monkey could achieve a higher coverage for FM Radio since it is possible to access most of its features from the main activity, which was not the case for Moto Help. This last one is an APK with several levels of screen navigations, and Monkey used to turn off the Wi-Fi, which blocked it during its exploration. For Digital TV and Camera, surprisingly, Monkey was not able to open both APKs properly. We discuss in more details these extra observations in Section 4.6.

4.2 EVALUATION #1: MONKEY'S CONTRIBUTION TO AETINGS' RESULTS

In this section, we perform an evaluation about the effect of Monkey usage on the overall AETings' results in the dependent variable *Coverage of the TBC set*.

In the experiment presented in Section 4.1, we shared the Coverage of the TBC set of AETing for the Motorola APKs. Since AETing strategy covers two missions (Recall from Section 3.1), perform the ArcWizad's test scenario last step suggestion and release Monkey in the source screen of the ArcWizad's test scenario last step, we want to know how much of the coverage result is the contribution of Monkey.

To attain the investigation of this section, we executed a modified version of AETing where it does not release the Monkey. Instead, this modified version only executed the ArcWizards' suggestions. Then, we compare the results with the results achieved by AETing original implementation during the experiment of Section 4.1.

Although in this section we are not conducting a formal experiment, we executed the modified AETing following the same protocol we established in the experiment of Section 4.1.3. In summary, the modified AETing was executed 20 times for each Motorola APK. Also, before each execution, we cleaned up the APK cache and previous data to start a completely new execution without interference from previous runs.

The Table 6 shows the Coverage of the TBC set results for each Motorola APK for the

	FM Radio			Digital TV			Camera			Moto Help		
	TBC: 143			TBC: 477			TBC: 271			TBC: 142		
	AETing	AETing*	*Diff*	AETing	AETing*	*Diff*	AETing	AETing*	*Diff*	AETing	AETing*	*Diff*
1	104	95	*9*	325	217	*108*	122	116	*6*	63	66	*-3*
2	103	94	*9*	321	197	*124*	124	116	*8*	59	65	*-6*
3	111	94	*17*	335	197	*138*	131	115	*16*	59	67	*-8*
4	111	94	*17*	335	217	*118*	122	116	*6*	65	68	*-3*
5	104	94	*10*	310	220	*90*	124	115	*9*	68	65	*3*
6	97	94	*3*	332	194	*138*	122	116	*6*	67	66	*1*
7	107	94	*13*	320	271	*49*	120	115	*5*	70	67	*3*
8	103	94	*9*	309	217	*92*	124	116	*8*	73	68	*5*
9	104	94	*10*	328	272	*56*	129	116	*13*	70	65	*5*
10	104	94	*10*	328	198	*130*	122	116	*6*	68	65	*3*
11	104	95	*9*	330	191	*139*	126	116	*10*	72	66	*6*
12	103	94	*9*	307	217	*90*	127	116	*11*	62	65	*-3*
13	99	94	*5*	311	217	*94*	125	116	*9*	62	66	*-4*
14	110	94	*16*	329	277	*52*	126	116	*10*	68	64	*4*
15	111	94	*17*	316	191	*125*	126	116	*10*	65	68	*-3*
16	103	94	*9*	324	269	*55*	126	116	*10*	67	67	*0*
17	110	94	*16*	319	195	*124*	126	116	*10*	65	65	*0*
18	103	94	*9*	328	193	*135*	125	116	*9*	63	67	*-4*
19	111	94	*17*	320	194	*126*	123	116	*7*	68	66	*2*
20	97	94	*3*	338	193	*145*	124	116	*8*	64	65	*-1*

Table 6 – Differences in Coverage of the TBC set between AETing (original implementation) and AETing* (modified implementation that does not execute Monkey).

modified version of AETing, which runs only the ArcWizards' suggestions. The column *Diff* is the difference of each execution between the original AETing and the modified one. We summarize the evaluation as follows:

- **FM Radio**: the coverage decreased on average in 10.85 methods (6.29%), in a minimum of 3 methods (2.1%), and a maximum of 17 methods (11.89%). It indicates that, in the original implementation of AETing, Monkey contributed with less than 12% of the total TBC coverage;
- **Digital TV**: the coverage decreased on average in 106.4 methods (22.31%), in a minimum of 49 methods (10.27%), and a maximum of 145 methods (30.4%). It indicates that, in the original implementation of AETing, Monkey contributed with less than 31% of the total TBC coverage;
- **Camera**: the coverage decreased on average in 8.85 methods (3.27%), in a minimum of 5 methods (1.85%), and a maximum of 16 methods (5.9%). It indicates that, in the

original implementation of AETing, Monkey contributed with less than 6% of the total TBC coverage;

- **Moto Help**: the coverage increased on average in 0.15 methods (0.11%) and in a minimum of 8 methods (5.63%). Plus, the coverage decreased a maximum of 6 methods (4.23%). It indicates that, in the original implementation of AETing, Monkey contributed positively with less than 5% of TBC coverage, but it also spoiled the TBC coverage by around 5%.

In overall, Monkey contribute with less than 30% in the Coverage of the TBC set. The highest Monkey's contribution was in Digital TV (around 30%), an APK that Monkey, when executing alone, could not even open. Similar to FM Radio, Digital TV is also an APK that most of its features is accessible from the main screen. It benefited Monkey when executed through AETing. The same happened for Camera, although the contribution was less than 6%. For these two APKs, we notice that Monkey is benefited when participating in AETing's strategy, since AETing handles the screen navigation and releases Monkey in the right place. The same is applicable for FM Radio, but with less contribution (maximum of 11.89%). For Moto Help we observed what we mentioned in the experiment from Section 4.1, where Monkey used to drive the device to a state that blocks the coverage. When AETing executed without Monkey, it did not suffer the same negative interference. AETing alone loss in maximum of 4.23% of coverage for not calling Monkey. Lastly, the Table 6 also shows that, when AETing executes alone, its coverage results are more stable than when Monkey participates in the strategy. Monkey naturally introduces a variation (positive or negative) due its pseudo-random events.

4.3 EVALUATION #2: EFFECTIVENESS IN GENERATING AUTOMATED TEST CASES

In this section, we evaluate the effectiveness of AETing in generating automated test cases from ArcWizards' test scenarios. For each set of test scenarios, AETing generated a corresponding set of automated test cases. We want to know what contributes to the capacity of translating the static analysis input (the test scenarios) into real code implementation (the automated test cases generated). Table 7 presents the data regarding the automated test case generation phase.

	FM Radio	Digital TV	Camera	Moto Help
Test Scenarios	29	25	22	102
Automated Test Cases	24	28	20	19

Table 7 – Effectiveness in generating automated test cases.

- **FM Radio**: ArcWizard generated 29 test scenarios for FM Radio. AETing generated 24 (82.76%) unique automated test cases from those of 29 test scenarios. AETing detected 1 test scenario not supported since it was suggesting a screen that is not mapped in Page Browser. After automated test case generation, it was discarded 4 automated test cases due to duplication;

- **Digital TV**: ArcWizard generated 25 test scenarios for Digital TV. AETing generated 28 (112%) unique automated test cases from those 25 test scenarios. The reason for the number of scripts generated surpasses the number of test scenarios is related to the multiple page objects that were found for some test scenarios (Recall from Section 3.2). For instance, a single test scenario suggested a screen that AETing found multiples page objects that maps it. Thus, for each page object, AETing generated an automated test case;

- **Camera**: ArcWizard generated 22 test scenarios for Camera. AETing generated 20 (90.91%) unique automated test cases from those 22 test scenarios. Two test scenarios were not supported by Page Browser since one indicates a widget that was not mapped in any page object and another that indicates a screen also not mapped. Five automated test cases were generated without AETing be able to trace a route to the last step source screen (Recall from Section 3.3.3);

- **Moto Help**: ArcWizard generated 102 test scenarios for Moto Help. AETing generated 19 (18.63%) unique automated test cases from those 102 test scenarios. Most of the test scenarios AETing received had the same last step. The higher number of the test scenario, in this particular case, was because the test scenarios suggested alternatives to the experienced human tester to reach the same goal. Since AETing only takes into account the last step, it generated several duplicates automated test cases that were discarded, remaining only the unique. One test scenario also suggested a screen that was not mapped into page objects and other test scenario suggested widgets not mapped as well. These not supported test scenarios were not converted into automated test cases.

4.4 EVALUATION #3: AETING AGAINST EXPERIENCED HUMAN TESTER

In this section, we compare AETings' results against an Experienced human tester (in short, Human in this section). We evaluated if AETing can cover the Coverage of the TBC set as much as a Human can following the same set of ArcWizards' test scenarios.

For the execution with the Human, it was set a one-hour exploratory testing session for each Motorola APK. During the sessions, the Human followed the ArcWizards' suggestions freely. He also used his knowledge to perform actions beyond the ArcWizards' suggestions, changing the system state, exploring the GUI elements, and other operations to increase the Coverage of the TBC set. Such a previous experience is considered the main factor of relevance for exploratory testing (Itkonen; Mäntylä; Lassenius, 2013).

Just a reminder about AETing executions. AETing was restricted to follow the test scenario suggestions implemented in the automated test cases generated. In an attempt to mimic in part the Human, AETing uses Monkey, but only at the last step source screen. AETing execution sessions lasted just the amount of time to complete the automated test case executions.

	FM Radio			**Digital TV**			**Camera**			**Moto Help**		
	TBC: 143			TBC: 477			TBC: 271			TBC: 142		
	AETing	Human	*Diff*	AETing	Human	*Diff*	AETing	Human	*Diff*	AETing	Human	*Diff*
Max	111	112	*-1*	338	382	*-44*	131	183	*-52*	73	76	*-3*
Min	97		*-15*	307		*-75*	120		*-63*	59		*-17*

Table 8 – The maximum and minimum Coverage of the TBC set for AETing from Table 3 along with the Coverage of the TBC set of the Human.

- **FM Radio**: the best result of AETing reached 111 (77.62%) and the worst one is 97 methods (67.83%), whereas Human covered 112 methods (78.32%). The biggest difference in coverage between AETing and Human is 15 methods (10%) and the shortest is 1 method (0.7%);

- **Digital TV**: the best result of AETing reached 338 (70.86%) and the worst one is 307 methods (64.36%), whereas Human covered 382 methods (80.08%). The biggest difference in coverage between AETing and Human is 75 methods (15.72%) and the shortest is 44 method (9.22%);

- **Camera**: the best result of AETing reached 131 (48.34%) and the worst one is 120 methods (44.28%), whereas Human covered 183 methods (67.53%). The biggest difference in coverage between AETing and Human is 63 methods (23.25%) and the shortest is 52 method (19.19%);

- **Moto Help**: the best result of AETing reached 73 (51.41%) and the worst one is 59 methods (41.55%), whereas Human covered 76 methods (53.52%). The biggest difference in coverage between AETing and Human is 17 methods (11.97%) and the shortest is 3 method (2.11%);

AETing reached a Coverage of the TBC set not more different than 15% to the Human results for FM Radio, Digital TV, and Moto Help. However, for Camera the data indicates that the difference is bigger than 20%. This difference was bigger here since Camera has several hidden GUI elements, which made AETing succeeds in the testing. The Human has the advantage in this particular scenario, since it can modify the system state in order to hit the target and increase the coverage. We discuss in detail AETing executions observations in Section 4.6. Table 8 summarizes the results we just described.

4.5 SUMMARY

In our empirical study we aimed at evaluating the following topics:

- Compare directly AETing against Monkey to identify which one is more efficient in covering the TBC;

- Measure the amount of Coverage of the TBC set contribution Monkey does increment in AETing's result;

- AETing's efficiency in generating automated test cases from ArcWizard test scenarios using Page Browser resources;

- Identify whether or not AETing can achieve Coverage of the TBC set result close to Human experienced testers;

The results presented in Table 9 indicates that AETing is effective in generating automated test cases, although this effectiveness depends on the available resources. For instance,

	FM Radio TBC: 143	**Digital TV** TBC: 477	**Camera** TBC: 271	**Moto Help** TBC: 142
Test Scenarios	29	25	22	102
Automated Test Cases	24	28	20	19
AETing's TBC coverage	111	338	131	73
AETing without Monkey's TBC coverage	95	277	116	68
Monkey's TBC coverage	102	6	2	54
Human's TBC coverage	112	382	183	76

Table 9 – Resume of the outcome from the evaluations. The Coverage of the TBC set presented here is the best result of each testing approach.

ArcWizards' suggestions must be supported by the device under test, which we do not know previously due to the static nature of the test scenarios, Page Browser should have page objects that map the screens suggested by ArcWizard. Those results also show that AETing's results depend more on the available resources (ArcWizard's suggestions and Page Browser implementations) than Monkey, which also affected some executions negatively. However, the overall results of the combination of Monkey capabilities in AETing make clear that such a combination is the best approach for AETing. Beyond that, Monkey alone suffers without AETing's help to drive it to the screen should it be released. Lastly, AETing was able to get closer to the experienced human testers. We discuss deeply the results achieved in Section 4.7.

4.6 AETING'S BEHAVIOR

In the previous sections, we presented the experiment that test AETing and Monkey to identify which one is better when covering TBC (Section 4.1), we evaluate the amount of contribution Monkey aggregates to AETing's strategy (Section 4.2), we evaluate the AETing's effectiveness in generating automated test cases (Section 4.3), and finally we compared AETing against experienced human testers when following ArcWizard's test scenarios suggestions (Section 4.4). During those evaluations, we made observations regarding AETing's behavior in the automated test case generation and execution phases, which we detail in this section to better present AETing's capabilities.

4.6.1 FM Radio

During the execution of the 24 unique automated test cases generated by AETing, 9 automated test cases finished successfully and 15 failed. These 15 failed scripts required the following states in the APK to succeed:

- The scanning for radio frequencies must be happening;
- The recording of a radio show must be happening;
- *OK Google* voice feature must be set;
- Favorite tab must be active;
- Found radio frequencies tab must be active;
- Dark theme must be active.

The navigation that takes the device to the main activity succeeded, but not the navigation that takes the device to the last step source screen. It means, for these cases, the Monkey was performed on a different screen, instead of in the target one. The complete execution took 46 minutes, 14 minutes shorter than the experienced human tester (Figure 38).

During the Monkey standalone executions, it reached a Coverage of the TBC set close to AETing. FM Radio is an Android application that has not many screens to navigate, and most of the features are accessible through its main activity. These two characteristics benefit Monkey (Figure 39), since the main activity was the main target of its pseudo-random events, covering a higher number of TBCs' methods.

4.6.2 Digital TV

During the execution of the 28 unique automated test cases generated by AETing, 15 automated test cases finished successfully and 13 failed. These 13 failed test cases required the following states in the APK to succeed:

- The Globo Play channel TV signal must be available;
- The menu option which allows the opening of Settings must be visible;

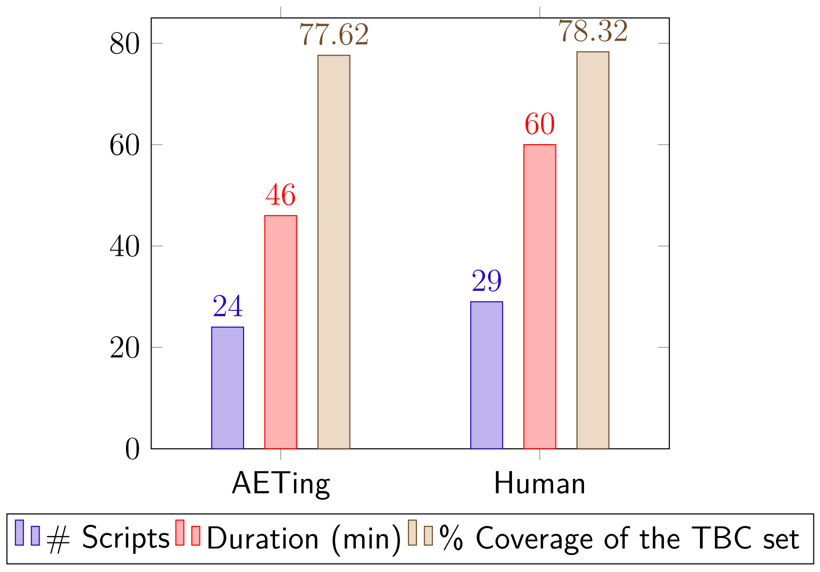


Figure 38 – Comparison between AETing and experienced human tester for FM Radio.

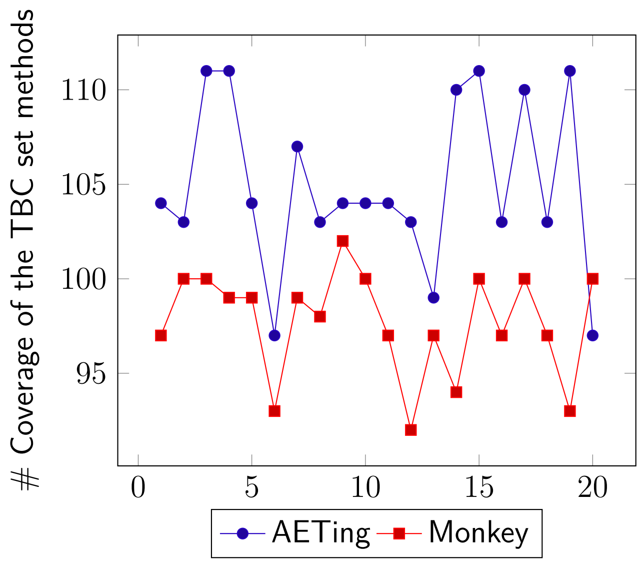


Figure 39 – Coverage of the TBC set results of 20 executions of AETing and Monkey. The TBC complete set for FM Radio was 143.

- A TV show recording must be ongoing;
- The warning raised when we first try to add a schedule must be accepted.

For these 13 failed automated test cases, the navigation that takes the device to the main activity succeeded, but it failed to navigate to the last step source screen. Thus, for these 13 automated test cases, the Monkey was performed on a different screen instead of in the target one. The complete execution took 92 minutes, 32 minutes longer than the experienced human tester (Figure 40).

When executing Monkey alone and it opened the Digital TV, a dialog was raised and Monkey did not dismiss it. Monkey got stuck in the screen that appears before the main activity during the entire execution. In other words, Monkey was not able to navigate to the main activity of the Digital TV properly. When AETing runs, it starts the screen navigations before releasing Monkey. Due to this, Monkey contributed more when released by AETing than when running by itself (Figure 41).

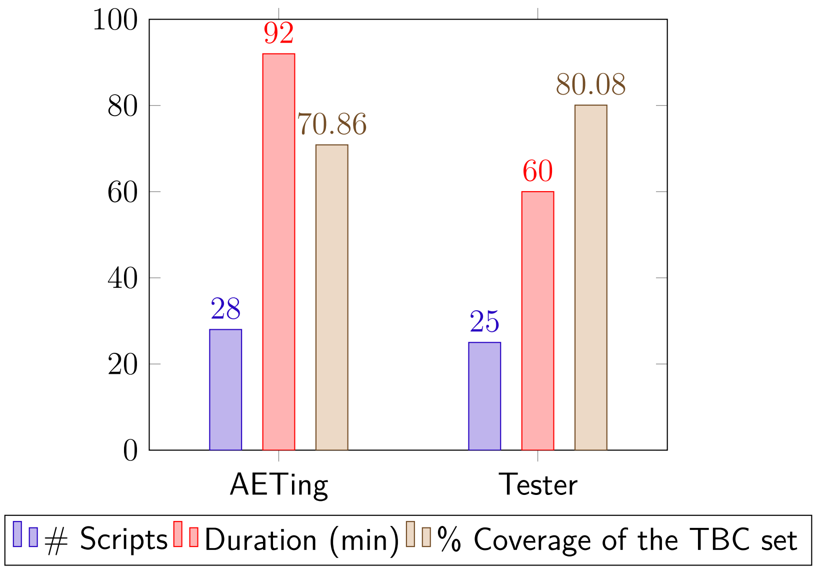


Figure 40 – Comparison between AETing and a experienced human tester for Digital TV.

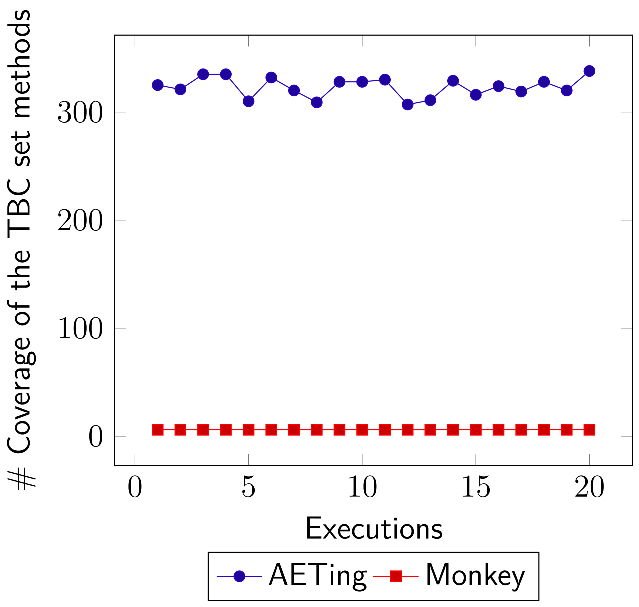


Figure 41 – Coverage of the TBC set results of 20 executions of AETing and Monkey. The TBC complete set for Digital TV was 477.

4.6.3 Camera

During the execution of the 20 unique automated test cases generated by AETing, 4 automated test cases succeed and 16 failed. Following the root causes for the failures:

- GUI elements mapped on page objects that were not visible on the screen;
- Page Browser failed to navigate to *Help* screen;
- Page Browser failed to navigate to *Cinemagraph* screen;
- Page Browser failed to navigate to *Cutout* mode;
- 5 automated test cases failed due to a lack of a route to the last step source screen.

The Camera has most of the GUI elements hidden from the user. To make them visible the user has to perform gestures on the GUI. Although these GUI elements are mapped into the page objects used in the automated test cases, they are not directly accessible. This lack of state drove 16 automated test cases to failure, being waiting for elements that are not accessible, which increased the execution time as well. Nevertheless, Monkey was executed at the screen the automated test cases could reach. The execution took 118 minutes, 58 minutes longer than the experienced human tester (Figure 42).

When triggering the standalone Monkey execution, it tried to open the Camera but the main activity closed before opening completely showing a toast notification saying *Device not supported*. Monkey was not able to open the Camera properly and spent 99 minutes playing around on the home screen but without opening the Camera again. It was a similar case to Digital TV (Section 4.6.2) where Monkey alone was inefficient. It could help AETing in its execution since AETing first starts its screen navigations before releasing Monkey (Figure 45).

4.6.4 Moto Help

During the execution of the 19 unique automated test cases generated by AETing, 1 automated test case succeeded, and the others 14 failed. Following the reasons for the failures:

- Page Browser dismissed GUI elements required by some automated test cases;
- No *Service center* button is listed at main activity screen, as mapped on page object;

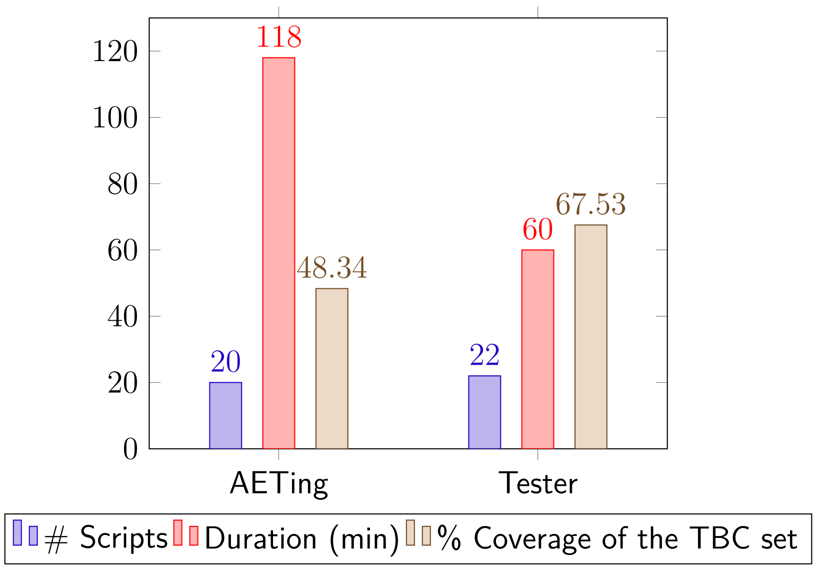


Figure 42 – Comparison between AETing and experienced human tester for Camera.

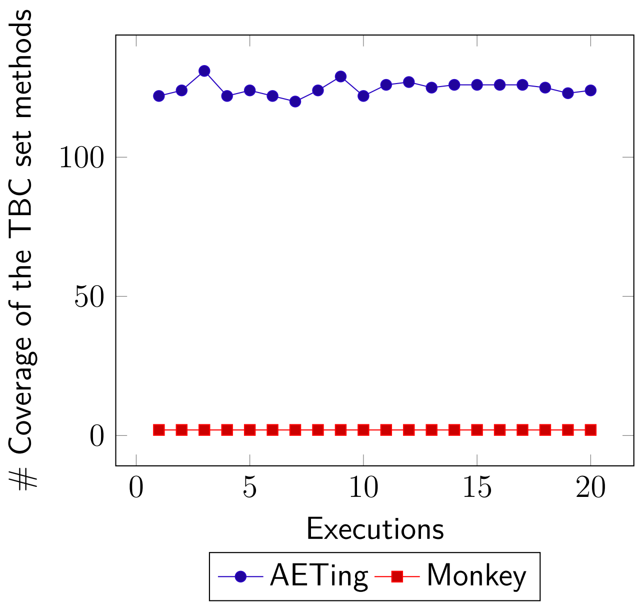


Figure 43 – Coverage of the TBC set results of 20 executions of AETing and Monkey. The TBC complete set for Camera was 271.

- Lack of correct state to make visible the button *Contact to us* at the hardware test result screen. It is only accessible when a hardware test is marked as failed;
- 3 automated test cases were generated from a test scenario that any route was found;
- 2 automated test cases were generated without a widget target of an action;
- The *RETRY* button at the *User Forum* screen is visible only when the device is offline.

For these 14 failed automated test cases we observed the initial setup of the device, which enabled the internet connection, conflicted with the automated test cases that required the device to be offline. Plus, although Page Browser handled the APK setup that appears on the first opening automatically, it also performed actions on the GUI that dismissed target widgets. Monkey also negatively affected AETing's results when it turned off the Wi-Fi connection, causing the failure of some automated test cases. The execution took 59 minutes, 1 minute less than the experienced human tester (Figure 44).

Monkey alone performed better for Moto Help in comparison to Digital TV and Camera. However, due to its random nature, it did not achieve a stable Coverage of the TBC set. During the multiple runs, two moments drove Monkey to a lower Coverage of the TBC set: (1) when the Wi-Fi was turned off, which blocked the access to some screens during most part of the Monkey's execution; (2) when Monkey navigated into *Open source licenses* screen and it did not navigate back, getting stuck (Figure 39).

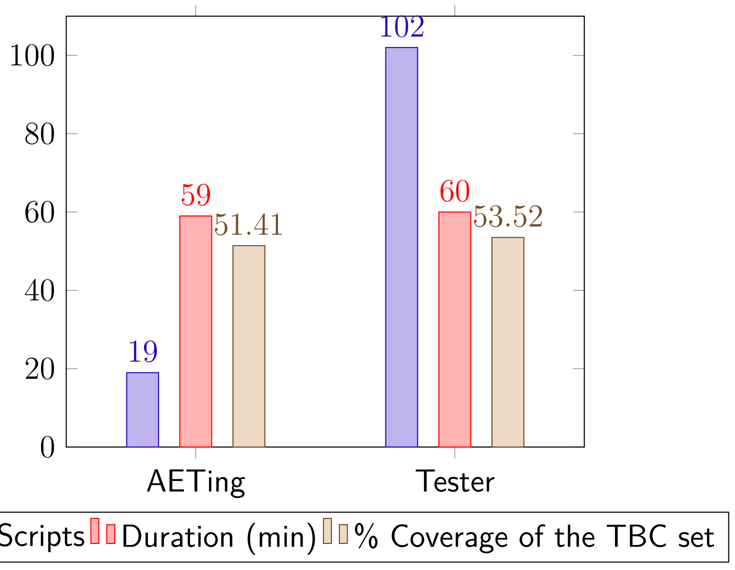


Figure 44 – Comparison between AETing and experienced human tester for Moto Help.

4.6.5 FDR execution time

During AETing's automated test case generation phase, it uses FDR to trace routes to the ArcWizard's test scenario last step source screen. Every FDR call performed by AETing took less than one second to find a route to the target screen.

Following the numbers of automated test cases per APK: 24 for FM Radio, 28 for Digital TV, 20 for Camera, 19 for Moto Help. In total, 91 automated test cases. The mean time speed of FDR for tracing a route for these 91 automated test cases is 0.7 seconds.

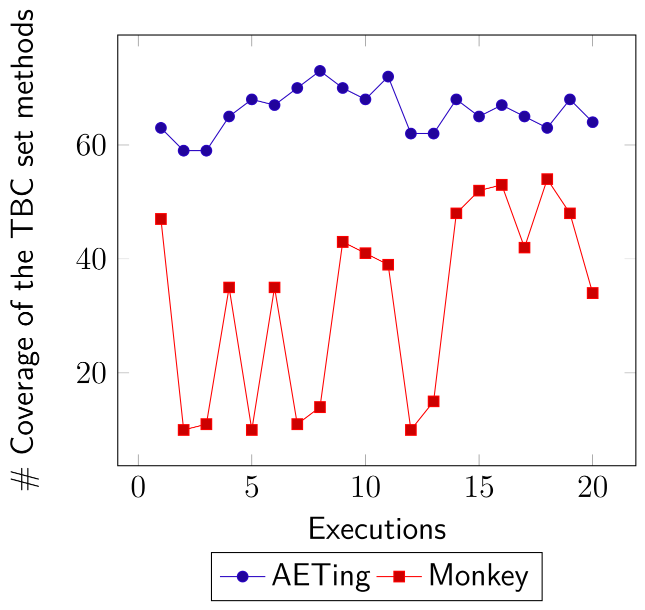


Figure 45 – Coverage of the TBC set results of 20 executions of AETing and Monkey. The TBC complete set for Moto Help was 142.

4.7 AETING'S BEHAVIOR LIMITATIONS

In the following subsections, we detail elements observed during our empirical study that affected AETing's behavior.

4.7.1 FM Radio

FM Radio uses Android Fragments to separate three screens under the main activity. These screens are about the radio scan, radios scanned, and favorites. They are mapped under the page object that refers to the main activity and there is no route set to navigate to each of them separately. Because of this, we faced failures in the automated test cases that tried to access the favorite tab and radios frequencies saved. FM Radio also uses *OK Google* to allow voice interaction with the APK, but it requires a human voice to be set in the device. The other two features are the *Scanning Radios* and the *Radio Show Recording*. These three features required special conditions for the device to be able to navigate into their specific screens. The same can be said for the enabling of the dark theme. The test scenarios suggested these screens as targets and they are mapped on Page Browser. However, on both sides (ArcWizard and Page Browser) there is no information about what must be done before making them accessible.

4.7.2 Digital TV

Digital TV has several buttons hidden at the video player, which only appears for a short period after a quick tap over the player region. When trying to interact with them, the automated test cases increased the execution time since the Page Browser waits a time to have them visible. As they are hidden, the automated test cases failed with a long execution duration. It is another situation where the state condition in accessing a GUI resource affects the execution. Most of the automated test cases use a route traced through the card *Globo Play* to access the detailed screen. This TV channel was not available in the region where the device was during execution. Plus, the available TV channel was not even mapped in Page Browser. However, if the available TV channel were mapped, FDR would suggest one route or another, since both are paths to a common destination. In this case, since AETing has no state information, it would not be able to help FDR during tracing. There is a screen where it is possible to schedule a TV show to be recorded, but it raises a pop-up before the input fields are accessible. Without dismissing this pop-up, the automated test case can not interact with it. Digital TV was also the first APK that Monkey alone could not cover very well. After opening the Digital TV, a pop-up message appeared and Monkey dismissed it. Then, Monkey got stuck at the screen that was behind the dismissed pop-up and could not move forward or backward. It released all its pseudo-random events, but none of them moved the APK to the next screen (the main activity). Still, the Digital TV APK was not frozen on the screen Monkey got stuck though.

4.7.3 Camera

Camera GUI elements are under a single Android Fragment (similar to FM Radio), and most of its GUI elements are also hidden (similar to Digital TV). Page Browser had problems navigating to some screens. When trying to navigate to *Help* screen, *Cinemagraph* screen and *Cutout* mode the issue was the same. Taking the *Cutout* mode as an example, this option is on the screen mapped in Page Browser as `Modes`. However, Page Browser could not go from the main screen to `Modes`. As a consequence, any option from that screen, which they were the target of the failed automated test cases, was not reachable. Monkey, alone, could not cover the Camera as well. The Camera did not even start. It closed right after the attempt of Monkey to open it. The Monkey could not do what AETing did to start the test: navigate to the main activity screen.

4.7.4 Moto Help

Moto Help is an APK that depends on several system states in order to enable/disable specific features or screens. During the executions, we observed that some automated test cases required the device to not have an internet connection. It conflicted with the device's initial setup, where we connected the device to the internet through Wi-Fi. However, others required an internet connection. We could not set up the device for both contexts. Here, the lack of system state knowledge, and control, blocked AETing to create more responsive automated test cases. Page Browser helped during the first opening of the Moto Help, where a wizard setup appears. This behavior is already implemented into the framework and is triggered automatically when using `.open()` API (Recall from Section 2.3). In other moments, Page Browser also automatically dismissed target widgets, which drove the automated test cases to fail. These circumstances appear because AETing has no control over the system state through Page Browser, at the same time it also does not know the proper requirements to perform navigation and actions suggested by ArcWizard, due to its stateless nature.

4.7.5 Available resources usage

By using ArcWizard's test scenarios and Page Browser infrastructure AETing inherits their limitations. From the test scenarios perspective, AETing tries its best to follow the suggestions given, but without any clue if the suggestion is supported by the device under test and overall requirements needed to make such suggestions possible to be executed. Page Browser is the infrastructure that AETing uses to match the stateless ArcWizard suggestions with their corresponding page object implementations. However, as we could observe more clearly in Camera and Moto Help executions, once AETing uses Page Browser implementations, AETing has no control over the device. If Page Browser benefits AETing when handling the setup wizard of Moto Help, it also drives the device to a state that makes the automated test cases fail. The Camera is another example, where Page Browser was not able to drive the device to the expected screen. In fact, the current version of the implementation of Page Browser page objects for Camera also causes automated test case failures. Digital TV executions showed that the combination of stateless suggestion to guide FDR to trace a route based on Page

Browser page objects markup also suffer from the lack of system state knowledge. The routes traced that travel over the *Globo Play* card are completely valid. The markup on Page Browser is correct, and FDR could find the proper route as well. However, that specific card was not available during execution. Any other card would fit in the scenario, but AETing had no resource to predict and also to act properly to avoid that blocker. In summary, if ArcWizard is the lighthouse that gives the destination for AETing, Page Browser is the train where the machinist only turns on the vehicle to always follow the rails defined instead of conduct the travel. That is, Monkey is not taking a car ride with AETing, as we mentioned before. AETing is the machinist of the train that Monkey travels.

AETing could generate automated test cases from any ArcWizard test scenario. However, not all ArcWizard test scenarios were supported by Page Browser page objects. We observed this happening for screens and widgets suggestions that were not mapped into Page Browser. To succeed in the generation, AETing depends on the available resources. When a test scenario suggests a screen that is an Android Fragment, multiple automated test cases are generated, since the page objects that map those Android Fragments have the same window. Although FDR found routes based on the Page Browser, AETing could not decide among multiple results, and that was the case for Digital TV with the route that goes through the Globo Play card, a valid path that was not available due to system state. Despite current limitations, the automated test cases generated by AETing almost reached the Coverage of the TBC set achieved by experienced human testers. The difference was not higher than 20%. Still, we observed that Monkey alone could not overcome AETing results, which means it works better along AETing instead on its own. We could watch Monkey fails to open Camera APK or to handle the alert popup on the first opening of Digital TV. Indeed, Monkey does suffer in Coverage of the TBC set terms without AETing's help. Lastly, the Coverage of the TBC set achieved only performing the last step action suggested by the ArcWizard test scenario was closer to the total achieved when executing the default behavior of AETing. That is, the stateless suggestions of ArcWizard had an impact on AETing's behavior covering the TBC.

5 CONCLUSION

To create Android APKs with quality, testers use several techniques. Since it is still very expensive to maintain test cases, we developed AETing, a solution to handle changes based on source code analysis. AETing works as a bridge between the stateless screen navigation suggestions from ArcWizard and implementations of screen navigations in Page Browser as page objects. AETing creates automated test cases using page objects to cover code changes detected by ArcWizard. Its main strategy is to model Page Browser into CSP_M terms and ask FDR to trace a route to the target. This target is the suggestion of the ArcWizard generated test scenarios based on code change between two given APKs. Thus, we achieved our main contributions: (1) mapping page objects as CSP_M specification; (2) creating automated test cases from FDR refinement output. A possibility to cover those limitations is to integrate a solution that maps the system state automatically as DroidBot (LI et al., 2017). Despite AETing limitations, notably, the solutions offered by the Academy did not work during our evaluations.

During our experimentation, we observed that our solution depends on the quality of the ArcWizard input and Page Browser infrastructure. We had test scenarios suggesting interaction with elements that are not mapped on page objects. But it is hard to say if the suggestion is wrong or if the page object missed some needed implementation code, for instance. We could observe several other examples where the system state blocks AETing to reach a higher TBC coverage. These examples are related to the nature of the applications and also their interaction with Android ecosystem features, which AETing has no knowledge about.

The results showed that AETing suffers due to the lack of system state information to achieve a given target from ArcWizard. In addition, while the automated test cases run, Page Browser keeps enabled features to handle automatically some screens (e.g: pop-up permissions, setup of the first use of an APK). It interferes in the system state, with no choices to AETing. Although our solution could not overcome the lack of system state information, in fact, it manages the available resources to produce automated test cases to cover code changes, producing results closer to experienced human testers. AETing is fully automated and offers an advantage to overnight test suites runs. Our results impressed Motorola staff and this technology will be integrated into the real production testing process.

5.1 RELATED WORK

Automatic Android testing is a field that continues attracting researchers' attention to explore different techniques. There is plenty of effort in developing fully automated approaches that effectively exercise the Android with a lower cost and without human intervention. Many of them are based on models, be it static or dynamic, others in input generation. Below we resume some of the existing approaches that are related to AETing.

JPF-Android (MERWE; MERWE; VISSER, 2012) is a model-based approach that enables the automatic verification of APKs on the standard Java Virtual Machine (JVM) based on the JPF model checker. JPF-Android provides a simplified model of the Android framework implemented manually based on JPFs' extensions that allows JPF to run the APK out of its natural environment.

JPF-Mobile (KOHAN et al., 2017) is another model-based approach that delivers an APK that wraps an adapted version of JPF to the Android environment. This way, JPF-Mobile tries to overcome the limitation of JPF-Android regarding the manual Android environment modeling.

Concolic Android TEster (CATE) (McAfee; Wiem Mkaouer; Krutz, 2017) is a model-based approach that delivers JPF capabilities to the Android platform through concolic static analysis on APK source files. It re-builds the APK as a JAR file that holds the simulation of the Android environment through Roboletric testing framework.

EHBDroid (SONG; QIAN; HUANG, 2017) is an event handler-based approach. It takes advantage of the event-driven system characteristic of the Android platform to generate all sorts of GUI events. It interacts directly with the event handler of the GUI event through source code instrumentation instead of generating GUI inputs.

DroidWalker (HU; MA; HUANG, 2017) is a model-based approach that builds a dynamic-adaptive model manipulating the Android device to perform screen navigations and register the system state of such navigations as well. From the dynamic built model, we can select a given state to drive the Android device to the selected point.

DroidBot (LI et al., 2017) is a GUI input generator for Android based on the state transition model generated dynamically while performing screen navigations, similar to DroidWalker. It builds the APK model exploring states through the Android testing/debugging tools, which enables it to work virtually in any APK. The model built is used along with a script that holds the resources it can use to input into the correspondent GUI fields when browsing the APK.

In comparison with those tools, AETing is fully-automated and does not explore the APK state blindly, which is a timing consuming task. This is an important characteristic of AETing since it avoids the challenges of automatic screen navigation, such as recognizing the uniqueness of a screen despite its state and avoiding cyclical navigation. Instead, it re-uses ArcWizard knowledge, which promptly points the direction where AETing should go. AETing also takes advantage of the existing Motorola's testing framework, called Page Browser, that supports updated Android versions to control the device accordingly. The combination of ArcWizard suggestions along with the already implemented framework allows AETing to be ready for an industrial environment.

	JPF-Android	**JPF-Mobile**	**CATE**	**EHBDroid**	**DroidWalker**	**DroidBot**	**AETing**
Model-based	X	X	X		X	X	X
Event handler-based				X			
Manual Android modeling	X						
Android up-to-date support			X		X		X
GUI interaction	X		X	X	X	X	X
APK re-compiling		X	X	X	X		
Store system state					X	X	
Blind screen navigations	X	X	X		X	X	
Trace route to cover code evolution							X
Fully automated			X			X	X

Table 10 – The green lines are desirable features. The other lines are undesirable features. From the desirable features, AETing only does not support *Store system state*. Plus, AETing does not have any undesirable feature.

5.2 FUTURE WORK

The first future work is to overcome the limitations inherited by AETing due to the stateless nature of the ArcWizard test scenarios. As we could observe, there is no available information about the system state required to make the ArcWizard test scenario steps possible. To do this, we intend to integrate the system state information regarding the screen navigations through direct interaction with the Android device. It will allow the improvement of the CSP$_M$ model to FDR trace more assertive routes.

Secondly, another future work is to integrate AETing to a mechanism that allows free device manipulation, since AETing is limited to follow what is implemented in the Page Browser. This restriction of the device control drove AETing to generate code that took the device to a non desired state, making automated test cases fail.

The third future work is to implement an intelligence combining the system state information with the possibility of free device manipulation to correct and complete an existing navigational model, in our case the Page Browser. It will reduce the impact of outdated implementations in the Page Browser. Currently, AETing depends on Page Browser developers to maintain and expand such existing code baseline.

Finally, the last future work we intend to identify whether the changes registered in TBC is applicable for the connected device. It will avoid AETing to generate an automated test case that would not fit for the connected device and fail during testing execution.

REFERENCES

ALÉGROTH, E.; FELDT, R.; KOLSTRÖM, P. Maintenance of automated test suites in industry: An empirical study on visual GUI testing. *CoRR*, abs/1602.01226, 2016. Disponível em: <http://arxiv.org/abs/1602.01226>.

Android Activity Documentation. *Activity | Android Developers*. 2020. Disponível em: <https://developer.android.com/reference/android/app/Activity>.

Android Input Documentation. *Input events overview | Android Developers*. 2020. Disponível em: <https://developer.android.com/reference/android/app/Activity>.

BOURQUE, P.; FAIRLEY, R. E. (Ed.). *SWEBOK: Guide to the Software Engineering Body of Knowledge*. Version 3.0. Los Alamitos, CA: IEEE Computer Society, 2014. ISBN 978-0-7695-5166-1. Disponível em: <http://www.swebok.org/>.

CARVALHO, G.; BARROS, F.; CARVALHO, A.; CAVALCANTI, A.; MOTA, A.; SAMPAIO, A. Nat2test tool: From natural language requirements to test cases based on csp. In: *18th Brazilian Symposium on Formal Methods*. [S.l.]: Springer International Publishing, 2015. p. 283–290.

CRAMER, D.; HOWITT, D. The sage dictionary of statistics : a practical resource for students in the social sciences. In: . [S.l.: s.n.], 2004.

DAS, K. A brief review of tests for normality. *American Journal of Theoretical and Applied Statistics*, v. 5, p. 5, 01 2016.

FOWLER, M. *PageObject*. 2013. Disponível em: <https://martinfowler.com/bliki/PageObject.html>.

GIBSON-ROBINSON, T.; ARMSTRONG, P.; BOULGAKOV, A.; ROSCOE, A. FDR3 — A Modern Refinement Checker for CSP. In: ÁBRAHáM, E.; HAVELUND, K. (Ed.). *Tools and Algorithms for the Construction and Analysis of Systems*. [S.l.: s.n.], 2014. (Lecture Notes in Computer Science, v. 8413), p. 187–201.

GRECH, V.; CALLEJA, N. Wasp (write a scientific paper): Parametric vs. non-parametric tests. *Early Human Development*, v. 123, p. 48 – 49, 2018. ISSN 0378-3782. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0378378218302561>.

GROVE, D.; CHAMBERS, C. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.*, Association for Computing Machinery, New York, NY, USA, v. 23, n. 6, p. 685–746, nov. 2001. ISSN 0164-0925. Disponível em: <https://doi.org/10.1145/506315.506316>.

HECKERT, N. A.; FILLIBEN, J. J.; CROARKIN, C. M.; HEMBREE, B.; GUTHRIE, W. F.; TOBIAS, P.; PRINZ, J. NIST/SEMATECH e-Handbook of statistical methods. 2012.

Herbsleb, J. D.; Mockus, A. An empirical study of speed and communication in globally distributed software development. *IEEE Transactions on Software Engineering*, v. 29, n. 6, p. 481–494, 2003.

HOARE, C. A. R. *Communicating Sequential Processes*. USA: Prentice-Hall, Inc., 1985. ISBN 0131532715.

HU, Z.; MA, Y.; HUANG, Y. *DroidWalker: Generating Reproducible Test Cases via Automatic Exploration of Android Apps*. 2017.

Itkonen, J.; Mantyla, M. V.; Lassenius, C. Defect detection efficiency: Test case based vs. exploratory testing. In: *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*. [S.l.: s.n.], 2007. p. 61–70.

Itkonen, J.; Mäntylä, M. V.; Lassenius, C. The role of the tester's knowledge in exploratory software testing. *IEEE Transactions on Software Engineering*, v. 39, n. 5, p. 707–724, 2013.

Java Pathfinder Documentation. *Home · javapathfinder/jpf-core Wiki · GitHub*. 2020. Disponível em: <https://github.com/javapathfinder/jpf-core/wiki>.

JPF-Android Documentation. *JPF-Android*. 2017. Disponível em: <https://heila.bitbucket.io/jpf-android/>.

JURISTO, N.; MORENO, A. M. *Basics of Software Engineering Experimentation*. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2010. ISBN 1441950117.

KOHAN, A.; YAMAMOTO, M.; ARTHO, C.; YAMAGATA, Y.; MA, L.; HAGIYA, M.; TANABE, Y. Java pathfinder on android devices. *SIGSOFT Softw. Eng. Notes*, Association for Computing Machinery, New York, NY, USA, v. 41, n. 6, p. 1–5, jan. 2017. ISSN 0163-5948. Disponível em: <https://doi.org/10.1145/3011286.3011292>.

Korel, B.; Tahat, L. H.; Vaysburg, B. Model based regression test reduction using dependence analysis. In: *International Conference on Software Maintenance, 2002. Proceedings.* [S.l.: s.n.], 2002. p. 214–223.

LI, Y.; YANG, Z.; GUO, Y.; CHEN, X. Droidbot: A lightweight ui-guided test input generator for android. In: . [S.l.: s.n.], 2017. p. 23–26.

McAfee, P.; Wiem Mkaouer, M.; Krutz, D. E. Cate: Concolic android testing using java pathfinder for android applications. In: *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. [S.l.: s.n.], 2017. p. 213–214. ISSN null.

MERWE, H. van der; MERWE, B. van der; VISSER, W. Verifying android applications using java pathfinder. *ACM SIGSOFT Software Engineering Notes*, v. 37, p. 1–5, 2012.

Mirzaaghaei, M.; Pastore, F.; Pezze, M. Supporting test suite evolution through test case adaptation. In: *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. [S.l.: s.n.], 2012. p. 231–240.

Monkey Documentation. *UI/Application Exerciser Monkey | Android Developers*. 2019. Disponível em: <https://developer.android.com/studio/test/monkey>.

MYERS, G. J.; SANDLER, C.; BADGETT, T. *The Art of Software Testing*. 3rd. ed. [S.l.]: Wiley Publishing, 2011. ISBN 1118031962.

NACHAR, N. The mann-whitney u: A test for assessing whether two independent samples come from the same distribution. *Tutorials in Quantitative Methods for Psychology*, v. 4, 03 2008.

NOGUEIRA, S.; ARAUJO, H. L. S.; ARAUJO, R. B. S.; IYODA, J.; SAMPAIO, A. Automatic generation of test cases and test purposes from natural language. In: CORNÉLIO, M.; ROSCOE, B. (Ed.). *Formal Methods: Foundations and Applications*. Cham: Springer International Publishing, 2016. p. 145–161. ISBN 978-3-319-29473-5.

PRESSMAN, R. S. *Software Engineering: A Practitioner's Approach*. 6. ed. USA: McGraw-Hill, Inc., 2004. ISBN 007301933X.

REDDEN, J. D.; RYAN, F. A. *Filosofia da Educação*. 5th. ed. [S.l.: s.n.], 1973.

REIS, J.; MOTA, A. Aiding exploratory testing with pruned gui models. *Information Processing Letters*, p. 49–55, 03 2018.

ROSCOE, A. *The Theory and Practice of Concurrency*. [S.l.: s.n.], 2005.

ROSCOE, A. W. Modelling and verifying key-exchange protocols using csp and fdr. In: *Proceedings of the 8th IEEE Workshop on Computer Security Foundations*. USA: IEEE Computer Society, 1995. (CSFW '95), p. 98. ISBN 0818670339.

Ryder, B. G. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, SE-5, n. 3, p. 216–226, 1979.

SCHNEIDER, S. *Concurrent and Real Time Systems: the CSP approach*. 1st.. ed. New York, NY: John Wiley Sons, Ltd, 1999.

SHAH, S. M. A.; GENCEL, C.; ALVI, U. S.; PETERSEN, K. Towards a hybrid testing process unifying exploratory testing and scripted testing. *Journal of Software: Evolution and Process*, v. 26, n. 2, p. 220–250, 2014. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1621>.

SOMMERVILLE, I. *Software Engineering*. 9th. ed. USA: Addison-Wesley Publishing Company, 2010. ISBN 0137035152.

SONG, W.; QIAN, X.; HUANG, J. Ehbdroid: Beyond gui testing for android applications. In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. [S.l.]: IEEE Press, 2017. (ASE 2017), p. 27–37. ISBN 9781538626849.

STOCCO, A.; LEOTTA, M.; RICCA, F.; TONELLA, P. Apogen: automatic page object generator for web testing. *Software Quality Journal*, 08 2016.

VALLEE-RAI, R.; CO, P.; GAGNON, E.; HENDREN, L.; LAM, P.; SUNDARESAN, V. Soot - a java bytecode optimization framework. *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, 10 1999.

WOHLIN, C.; RUNESON, P.; HST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLN, A. *Experimentation in Software Engineering*. [S.l.]: Springer Publishing Company, Incorporated, 2012. ISBN 3642290434.