Master of Science in Software Engineering

September 2018

Comparing Different Approaches of GUI Testing for Mobile Applications on Android Platform

Yuhao Min, Shengcong Cai

Faculty of Computing, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfilment of the requirements for the degree of Master of Science in Software Engineering. The thesis is equivalent to 20 weeks of full time studies.

The authors declare that they are the sole authors of this thesis and that they have not used any sources other than those listed in the bibliography and identified as references. They further declare that they have not submitted this thesis at any other institution to obtain a degree.

Contact Information:

Author(s):

Yuhao Min

E-mail: yumi16@student.bth.se

Shengcong Cai

E-mail: shca16@student.bth.se

University advisor:

Bogdan Marculescu

Faculty of Computing

Blekinge Institute of Technology   Internet : www.bth.se

SE-371 79 Karlskrona, Sweden   Phone : +46 455 38 50 00

ABSTRACT

Background. With the development and popularization of mobile Internet, smartphones are becoming more and more popular. Android is one of the most popular platforms of smartphones. And application is one of the most important part of a smartphone. There are a lot of money and human resources spent on Android application development every year. And quiet a big part of them goes to quality assurance of applications. Graphic user interface (GUI) testing is one important part of its quality assurance. Android phones use touch screen as the major I/O method. Therefore, GUI testing on android platform shall be different to conventional software applications that are designed to run on desktop environment.
Objectives. The aim of this research is to assess the performance of two GUI testing approaches (2nd vs 3rd generation) of automated UI testing in terms of testing Android applications. By assessing these approaches, we could hopefully get insights of their advantages and limitations for using them in the context of Android development. And this aim can be divided into three objectives, to compare the time spent on implementing test cases of each tool, to compare the time costed when executing test cases of each tool, to compare the number of defects found by each tool.
Methods. The research methodology we chose is controlled experiment. We have chosen UI Automator and Appium to represent 2nd generation GUI testing approach, EyeAutomate and SikuliX to represent 3rd generation GUI testing approach. We used each tool to implement and execute 120 test cases to compare them on the time spent on implementing test cases of each tool, the time costed when executing test cases of each tool, the number of real defects found by each tool, and the number of false positives found by each tool.
Results. Tools using 3rd generation GUI testing approach take less time to implement test cases than tools using 2nd generation GUI testing approach. And there is no specific pattern when comparing tools using 2nd and 3rd generation GUI testing approaches in terms of time cost on executing test cases. It is different between different test cases. Besides false positive alerts appear at a much higher frequency in tools using 3rd generation GUI testing approach than tools using 2nd generation GUI testing approach. While, real defects found by each tool are the same.
Conclusions. 3rd generation GUI testing approach is more efficient in terms of implementing test cases than 2nd generation GUI testing approach. But 3rd generation GUI testing approach finds much more false positives than 2nd generation approach. To decide if a defect alert is false positive or not requires human effort. In a long term, it may accumulate huge lost on human efforts. Therefore, to maintain test cases, 3rd generation approach consumes lots of human efforts.

Keywords: Android Testing, Visual GUI testing, Property-based GUI testing

CONTENTS

ABSTRACT ..... **III**

CONTENTS ..... **IV**

1 INTRODUCTION **1**

1.1 INTRODUCTION 1

1.2 AUTOMATED GUI-BASED SOFTWARE TESTING 2

1.2.1 The 1st generation 3

1.2.2 The 2nd generation 3

1.2.3 The 3rd generation 3

1.3 MOTIVATION 4

1.4 STRUCTURE OF THESIS 5

2 RELATED WORK **6**

3 RESEARCH QUESTIONS **8**

4 EXPERIMENT PLANNING **9**

4.1 REASONS OF SELECTING CONTROLLED EXPERIMENT 9

4.2 CONTEXT SELECTION 9

4.3 VARIABLES 9

4.4 HYPOTHESES 10

4.5 SUBJECTS SELECTION 11

4.6 TESTING TOOLS SELECTION 14

4.6.1 Tools Using 2nd Generation Approach 14

4.6.2 Tools for 3rd Generation Approach 15

4.7 TEST CASE DESIGN 16

4.8 DEFECTS INJECTION 19

5 EXPERIMENT OPERATION **20**

5.1 PREPARATION 20

5.2 EXECUTION 20

6 DATA ANALYSIS AND INTERPRETATION **22**

6.1 PERFORMANCE IN TERMS OF IMPLEMENTING TEST CASES 22

6.2 PERFORMANCE IN TERMS OF EXECUTING TEST CASES 24
6.3 DEFECTS FOUND BY DIFFERENT TOOLS 26
6.4 RESULTS INTERPRETATION 27
7 THREATS TO VALIDITY 28
8 CONCLUSION 29
8.1 LESSONS LEARNED 29
8.2 CONCLUSION 30
9 FUTURE WORK 31
REFERENCES 32

1 INTRODUCTION

1.1 Introduction

With the development and popularization of mobile Internet, smartphones are becoming more and more popular. In the mobile market, the operating system of smartphone mainly consists of Android and iOS. Android is a Linux-based mobile operating system developed by Google. Besides, it is open source. Google published the source code of Android platform to the public, anyone can copy, modify and use it without paying fees, but they need to obey the law and rule in the license terms and conditions [1]. It is used widely on smartphone, tablet computer and other mobile devices. Statistics show that Android has more than 80% market share in global mobile phone [2].

Specifically, Android platform consists of an operating system, middleware and key applications [3]. The architecture of Android platform is Software Stack which mainly consists of four parts: Application, Application Framework, Android Runtime and Libraries as well as Linux Kernel, see Figure 1-1. The underlying layer is developed by C language based on Linux kernel. It only has some fundamental functions such as threading and memory management. Android Runtime and Libraries layer is developed by C and C++, it includes Library and Virtual Machine [4]. Runtime can support Just-in-time and Ahead of Time compilation. It can also help to collect garbage. Application Framework simplifies the reuse of components, it can help programmers develop programs quickly. The top layer consists of kinds of applications, they are usually developed by Java.

Apps(applications) is one of the most important part of an Android phone. Statistics show, there are more than 3.8 million apps in Android platform [5]. Users can install applications from Google Play. Besides, they can also download the application's APK (Android application package) file and install it. The most important components for Android apps are the Activities, they manage the screen of UI of apps [6]. Every activity includes some views which are coded by Java or in xml file. Activities also define all the interactions between users and UI. Applications are written by Android software development kit (SDK), they are usually developed by Java language. Programmers can also combine Java with C/C++ when they develop. Besides, Kotlin Language can be used in applications development.

With growing types of applications, their functionalities are very diverse. And the competitions between different applications and different companies are becoming more and more intense. Which make some developers plan to reduce cost to get more profit. However, the quality of those low-cost apps is staggered. End users usually would not choose the low-quality applications, even it has many users in the beginning, the number of users will decrease rapidly as time goes by. So, it is significant to test apps sufficiently before they are released. The industry has put a lot effort on ensuring the quality of Android applications.

Testing is one of the most important phases for quality assurance in software development. Because it is the most efforts consuming phase in development and it is also closely related with the quality of software [7]. Good test methods can help detect more bugs and save costs. In the opposite, the improper method can miss many bugs as well as waste money. Therefore, developers desire to find a way to minimize the effort as well as to detect bugs comprehensively. Automated testing can help to improve efficiency and reduce cost. In other words, automated testing is a good way for applications test. There are many Automated testing tools available for Android application development, e.g., Monkey, Robotium, Monkey runner, Appium, UI Automator, Test Bird. etc.

Most mobile applications are equipped with Graphical User Interfaces (GUIs). GUIs are the means through which end user interacts with the system. They provide us a more direct and user-friendly experience while using the system. In this paper, we focus on GUI testing (Graphical user interface testing). GUI testing is very important for ensuring quality of product, because it is starting from the end user's point of view [8]. Users can invoke almost all android applications by GUI. There are two methods of GUI testing: manual testing and automated testing. For a long time, people used manual testing to

detect bugs. In manual testing, testers always need to execute plenty of regression testing. Testers would feel boring. Especially, testers would make mistakes easily if the software is complex. So manual GUI testing would spend plenty of time and effort and it is difficult to detect real faults [9]. Developers desire to automate GUI testing for minimizing the effort and detecting more bugs. Automated GUI testing can help to execute regression testing more conveniently. Running regression testing automatically can improve testing efficiency and reduce regression testing time. And it can help to liberate testers' labor. Besides, automated testing can guarantee the quality of the test because the process of testing didn't have negligence and errors.

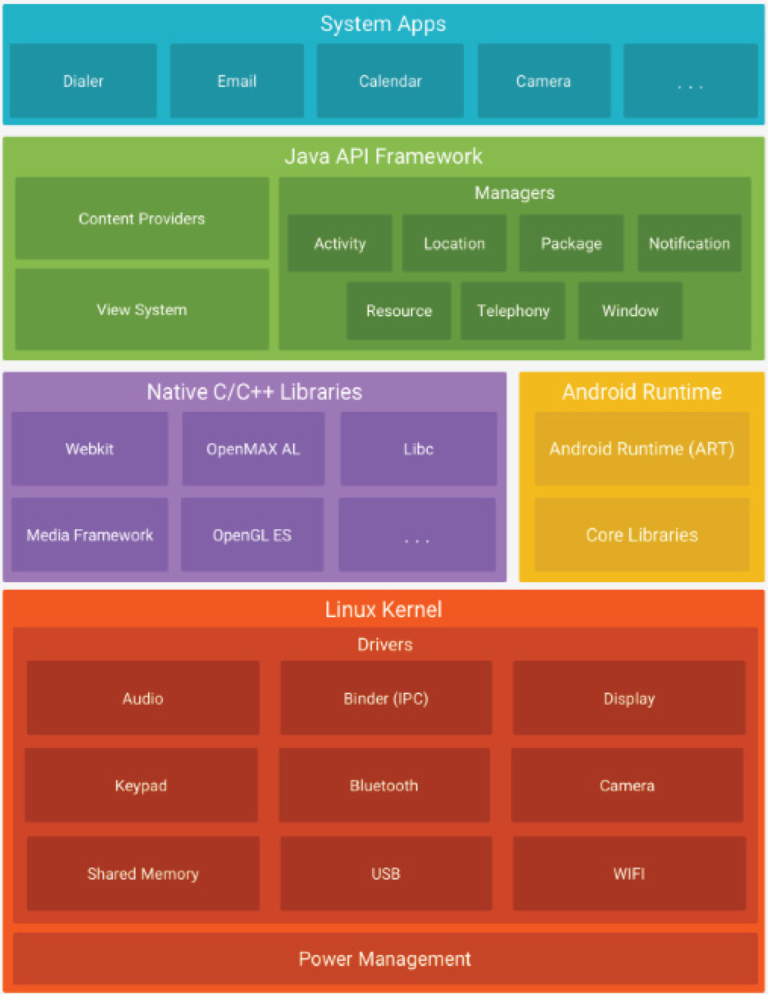

Figure 1-1 Android architecture

1.2 Automated GUI-based Software Testing

GUI testing is often considered at the level of system testing [6]. Previous research divided automated GUI testing into three generations based on approaches of how to identify GUI components. The 1st generation uses coordinates on screen to identify GUI components. The 2nd generation relies on GUI components and widgets that can be accessed through software artifacts (e.g., source code). And the 3rd generation uses image recognition technology to access the GUI components that are directly showed on the screen. It is also called Visual GUI-based testing (VGT) [10]–[13]. We would introduce these three generations detailed in the following text.

1.2.1 The 1st generation

The 1st generation GUI automated testing technique uses screen coordinates to interact with the SUT (system under test), testers can get these exact screen coordinates by recording manual interaction with the SUT [14]. Then, these coordinates would be saved into testing scripts for executing regression testing conveniently [15]. Testers can use these identified coordinates directly in next testing. This can help to improve the frequency of testing and reduce the effort. Here are some 1st generation GUI automated testing tools Abbot, Jacareto, JFCUnit, Marathon and Pounder. The problem of these tools is that the test case would fail if the layout of GUI changed slightly [16]. It leads to high maintenance cost. Especially, modern apps update fast. The 1st generation GUI automated testing technique is outdated quickly.

1.2.2 The 2nd generation

The 2nd generation GUI automated testing technique overcomes some challenges in the 1st generation. The 2nd generation GUI automated testing technique interact with SUT by using graphical user interface elements' properties (like index, text, resource-id, class, package, content-desc and so on) [10]. As it is shown in Figure1-2, testers can get the properties of GUI components by using UI Automate View. Then, the GUI components are located based on their properties. There are some 2nd generation GUI testing tools available for Android applications on the market: Monkey, UI Automator, Espresso, Robotism, Appium. When testers write test cases they could reuse the code if the properties of GUI component are same. Additionally, some 2nd generation GUI automated testing tools (like Appium) can make users use the same API against Mac, Windows and Linux OS. it means that users can reuse code on different platforms. This can help to improve efficiency and reusability. And most 2nd generation GUI testing tools have recording and replaying functions [15], this can help to reduce the cost of development.

On the other hand, there are some challenges. For example, some GUI components are generated in runtime. Before running the software, the properties of the components are unknow. These kinds of components cannot use this testing technique for testing. In addition, if the GUI components are changed (for example, the properties of the component are changed), testers need to modify the test cases which including these components. It leads to high maintenance cost in industrial practice. Second generation testing approach interacts with SUT' s internal workings. This is also a prerequisite for using these tools [14]. This technique does not verify the correctness of SUT's behavior and appearance from a graphical GUI point of view [15]. It resulted that this testing technique lacks flexibility in industrial practice.

1.2.3 The 3rd generation

The 3rd generation GUI automated testing technique uses image recognition technology to access the GUI components that are directly showed on the screen. It is a tool-driven technique, it interacts with SUT by image recognition [10], [17]. It is also called Visual GUI-based testing (VGT). Initial VGT was based on an automated tool (Triggers) which supported image recognition in 90s [18]. However, it was not used in industrial practice because of lacking hardware and image recognition technology support [19]. With the development of hardware and image recognition algorithm technology, there are some common VGT tools now: Sikuli, JAutomate, eggPlant, EyeAutomate and SikuliX.

VGT technique is similar to 2nd generation GUI automated testing technique but it uses image recognition to locate button, image, textbox and so on instead of using the property or code of GUI elements [19]. As it is shown in Figure1-3, after getting element we can perform multiple actions to it by static code information [11]. It mimics the interactions between human and SUT. It is easy to understand test cases of VGT because this process is visual.

VGT technique can be used on any system which is based on GUI conveniently. Because it can use image recognition technology to locate every element in the GUI. It can help to reduce effort observably comparing to write test scripts only by code. It has high flexibility. But there are many problems and challenges of VGT technique. For example, it is very dependent on image recognition technology, if image recognition occurred error test case would not work. And it is difficult to recognize smaller-sized images. Besides, test cases may not work when the size of image or layout of GUI changed. The reusability of VGT technique is low if software updates, it leads to high maintenance cost. And it is difficult to use VGT technique for testing non-GUI systems.

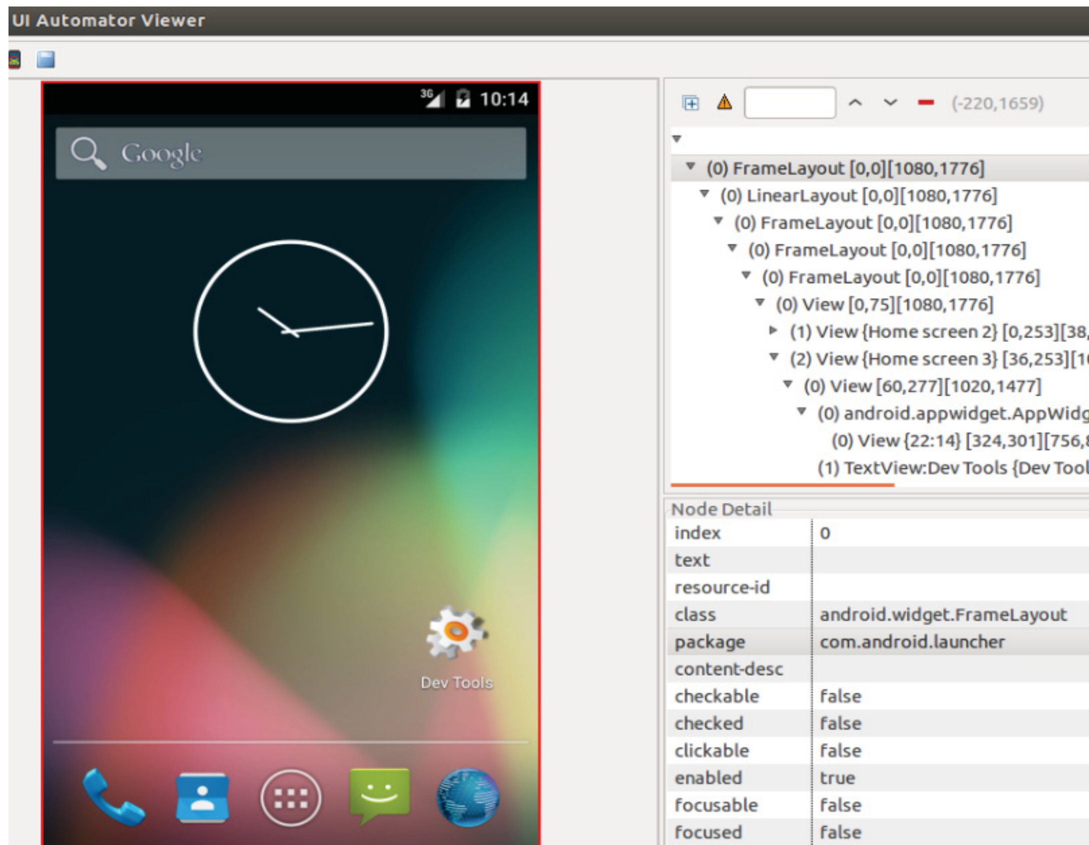

Figure 1-2

1.3 Motivation

The 1st generation GUI automated testing approach is not used independently in industrial practice because of the fragility and high maintenance cost. So, we do not investigate it. The 2nd generation GUI automated testing approach is the most widely used in GUI automatic testing. However, it still has some challenges. About 3rd generation GUI automated testing approach, VGT technique is still a new and immature technique, and it can overcome most of the challenges of previous techniques [14]. Although it is rarely used in industrial practice, it has high potential in GUI automated testing.

Previous research in VGT mainly focused on desktop or web applications [13], [15]. But few researches focused on applying VGT technique to mobile applications. Only a few companies desire to invest research and practice of VGT technique because they are not willing to investigate on those projects which are not profitable [10]. Conventional platform uses a dedicate monitor, a mouse and a keyboard as the basic input and output (I/O) method. While mobile platform mainly uses touch screen as the I/O method, which is very different to conventional one. So, the automated GUI testing on mobile platforms should also be s different story. In this project, we are going to find the feasibility of applying VGT to

Android application by comparing the performance of GUI testing tools which two tools use 2nd generation technology and the other two tools use 3rd generation technology in a controlled experiment.

1.4 Structure of Thesis

The structure of this paper are as follows. Section 2 presents the related work. Section 3 describes the aim and objectives of the study, as well as address the research questions. Section 4 describes how the experiment is planned. Its operation process is shown in section 5. Besides, we state the data analysis and its interpretation in section 6. In section 7, we analyze threats to its validity. In section 8, we provided the lessons learned and concluded the study. Some suggestions for future works were made in section 9.

2 RELATED WORK

We have found some papers about GUI automated testing techniques. We can initially get some advantages, disadvantages, challenges and limitations of the techniques from their research result. However, there is few researches of 3rd generation technique in android.

Siponen et al. [20] did a research about how to improve the maintainability of GUI automated testing. In this paper, authors used Visual GUI Testing technique for testing Android GUI mutations. Besides, they focus on regression testing by using VGT and 2nd generation techniques. They used 2nd and 3rd generation GUI testing approach to test the software application respectively. When the layout of the software or the property of a button is changed, they recorded the time spent on changing the test cases, then compared the data they collected. They found that VGT tools was more efficient than 2nd generation tools when testers modify the test cases. Besides, they found that VGT technique was more suitable than 2nd generation technique on system test and acceptance test.

Börjesson et al. [19] did a research to evaluates two popular VGT tools (Commercial Tool and Sikuli). They compared develop time, the size of the test cases and the execution times of test cases of two tools. They also tested animated and non-animated GUI software. The result shows that the performance of these tools is close. And VGT tools is very suitable for non-animated GUI software. Besides, two tools can also interact with animated GUI software. It can help to test many different types of software. But they also pointed out the challenges like high maintenance costs.

Septian et al. [20] evaluated 4 most used Android GUI testing tools: Espresso, Appium, Calabash and UI Automator. They developed an android application for testing. Although they did not prove which tools is the best, they got the result of tools' characterization and applicability after testing the application by using these four tools. For example, they found that Espresso was very suitable for basic and simple testing and it supported simple API. Espresso can be only used within the active applications. Appium supports many languages to write test scripts and also supports testing cross-app interactions. UI Automator can help to get properties of elements of GUI.

Alégroth et al. evaluated the applicability of VGT techniques in industrial practice [14]. After conducting some experiments of VGT in industry, they found that VGT technique can be used in industrial practice. Comparing to manual testing, runtime of test cases of VGT tools is faster and the frequency of testing is higher. VGT tools can find more bugs. And it can help to reduce testing time when testers execute regression testing. It is very flexible because it can be used on any system which is based on GUI. Additionally, the code of VGT tools is usually very simple, and VGT tools combines with visual operation. It is easy for testers to use VGT tools. Following the architecture of VGT scripts can help to write test cases. But they also pointed out some challenges and problems. For example, It is hard to use VGT tools on the system which is not based on GUI system. VGT tools are very dependent on image recognition technique, it is difficult to recognize smaller-sized images. Once the update of software changes the layout of GUI, the test cases should be modified. It leads to high maintenance cost. VGT techniques is not enough mature because of lacking research and practice. On other hand, they found conditions that help companies to use VGT as a long-term testing approach. Firstly, company should integrate VGT into testing. After writing test cases, testers should modify the code in time when software updated. Secondly, testers should follow the architecture of VGT scripts when they use VGT tools. Testers should write simple and clear test cases to reduce the complexity. It can help testers to modify test cases conveniently. Thirdly, testers should give priority to use stable test cases when they execute regression testing. It can help to reduce maintenance cost.

Alex et al. did a research about how to simplify the creation and maintenance of GUI testing [21]. In this paper, they introduced the reasons why it is difficult to conduct and maintain GUI testing and they gave some advice. They advised separating domain logic from UI by using the Model-View-Controller (MVC). It can help to find specific element of GUI. Testers should use multiple properties to locate a specific element that they want to test. Choosing an appropriate GUI tool which includes simple and

easy-to-use API can help testers conduct GUI testing. Besides, testers should modify test cases frequently.

Garousi et al. evaluated the performance of two VGT tools (Sikuli and JAutomate) [17]. In this paper, they found that these two tools both had 'Replay' problems. 'Replay' problems mean that the tools are difficult to recognize smaller-sized images. In addition, they found that Sikuli only supported Replay feature, and JAutomate supported both Record and Replay Features by comparing the playback functions of two tools.

Aho et al. did a research to improve GUI automated test modeling and support suitable tools [8]. In this paper they combined MBT (model-based testing) tool with GUI Driver tool. GUI Driver generates models, and these models are used to generate test sequences by using model-based testing technique. Then GUI Driver uses these test sequences to create test report.

Baek et al. created a set of multi-level GUI Comparison Criteria (GUICC) to address the problem of modeling for dynamic GUI [9]. Results indicate that generating a GUI model by multilevel GUICC is more efficient than using activity-based tools. In addition, GUI models generated by multilevel GUICC testing can ensure high code coverage in testing and it can detect more bugs.

Feldt et al. did a research about VGT techniques in industrial practice [10]. They interviewed Spotify and analyzed the experience and challenges of using VGT tools. The results indicate that Sikuli can be used for a long time in industry, but the company needs to innovate the development process to integrate VGT technique into developing. In addition, they also defined 14 guidelines to help companies use VGT tools.

By reading these related literatures, we have summarized some of the conclusions about GUI testing below:

1. Compared with 2nd generation testing technique, VGT technique is more suitable for system test and acceptance test
2. The test cases which are implemented using VGT tools is easier to modify than those implemented using 2nd generation testing tools.
3. Automated GUI testing can help to improve test frequency.
4. VGT can locate elements of GUI system easily, but it has robustness problems.
5. VGT has 'Replay' problems. It means that the VGT tools are difficult to recognize smaller-sized images.
6. VGT can be applied in industrial practice.
7. VGT tools and 2nd generation tools are very suitable for non-animated GUI software.
8. High maintenance cost of VGT
9. Automated GUI testing is suitable for non-animated GUI software, but it is not suitable for dynamic GUI.
10. The code of VGT tools is usually very simple, the code of 2nd generation tools are complex.

In this paper, our objective is to assess the performance of two GUI testing approaches (2nd vs 3rd generation) of automated UI testing in terms of testing Android applications. From related works, we found that most of the related literatures were focus on 2nd and 3rd generation GUI testing for web and desktop applications. Only few researches were based on the android, especially in the area of VGT technique.

In addition, we find that some of conclusions lack of experimental data from related work, such as the execution time of 2nd generation GUI test case is faster than the execution time of VGT test case. So, we assess the performance and limitation of two 2nd and 3rd generation automated GUI testing on Android.

3 RESEARCH QUESTIONS

The aim of this research is to assess the performance of these two approaches (2nd vs 3rd generation) of automated GUI testing in terms of testing Android applications. In previous research reports, the works they have done mostly focused on applying web-based applications and desktop applications. While few researches are done in applying VGT to Android development. Therefore, by conducting this research, we could hopefully get insights of their advantages and limitations for using VGT in the context of Android development comparing to the traditional GUI testing approaches.

Testing approaches could not be used to testing software applications directly. There are many testing tools available in the market that helps testers to implement these test approaches. And in this study, we can convert the aim of comparing these two testing approaches to comparing different testing tools that use these testing approaches. And to assure we are comparing between these approaches not just between different tools, we shall select multiple tools from each side. Details about tools selections are provided in section 4.3.

To assess the performance, there are two aspects that we should consider: *effectiveness,* i.e. how many defects are found by each tool, and *efficiency*, i.e. how much time are needed to implement and execute test cases.

By addressed these two aspects and selected four tools, the following objectives were formulated:

O1: Compare the time spent on implementing test cases of each tool.
O2: Compare the time needed when executing test cases of each tool.
O3: Compare the number of defects found by each tool.

O1 and **O2** are related to the efficiency aspect above, and **O3** is correspond to the effectiveness aspect of the tools.

After the objectives of the research is formulated, the following three research questions are addressed:

RQ1: How do the testing tools being considered compare with respect to the time costed of implementing test cases? From a software engineering point of view, it is important to take human related factors into consideration. And in this specific context, the most labor-intensive part is implementing test cases for each tool. While test cases are implemented by the same group of developers, we could convert the labor needed of implementing test cases to the time cost of implementing test cases. To measure this, we are going to record time spent on implementing test cases for each tool.

RQ2: How do the testing tools being considered compare with respect to the time needed when executing test cases? Another aspect we assess of these two approaches is the performance of the tool themselves in terms of the resources needed for executing test cases. While executing test cases in the same environment, i.e., all hardware and software environment is identical, the resource that are consumed could be regarded as the time spent on executing each test case.

RQ3: How do the testing tools being considered compare with respect to the number of defects found? When executing test cases for each approach, each tool provides a list of failed test cases that need to be reviewed. After that we need to decide whether they are a false positive (wrong warning that does not actually constitute a fault) or true positive (real fault). Thus, this research question can be divided into:

RQ3.1: How many false positives appear in each tool?
RQ3.2: How many real defects are found by each tool?

4 EXPERIMENT PLANNING

This section describes the planning of the experiment. The planning includes contest selection, hypothesis formulation, variables selection, selection of subjects, experiment design, instrumentation and validity evaluation of the experiment [22].

4.1 Reasons of Selecting Controlled Experiment

The research methodology we chose is controlled experiment. Because experiments usually involve more than one treatment while other settings or environment are controlled to be as identical as possible to compare consequences of different treatments [22]. And in this study, because we want to compare the performance of two different GUI testing tools in the context of Android development, an experiment provides us a way to control over different variables that might affect the performance of the tools, e.g., Android applications, testers, test cases, etc., and to focus on the only factor, i.e. GUI testing approaches, we want to investigate in this research.

Other research methodologies we could chose are survey and case study. Reasons for rejecting these two research methodologies are given in below.

A survey is often conducted when, for example, a tool or technique, has been in use for a while. And it is often performed in retrospect [22]. It is not suitable for in the specific context of this study. Because VGT is a new technology and has not been widely adapted in software development industry. Therefore, if we would conduct a survey through interview or questionnaire, we may not able to collect adequate data because of the small population of testers using VGT in Android development. And a survey is not that efficient in comparing different treatments of one factor.

Case study is more suitable for investigating one instance (or a small number of instances) of a contemporary software engineering phenomenon within its real-life context [22]. But the limited resources we have do not allow us to have access to real-life project of using VGT as the GUI testing approach in Android development projects. Because VGT is a new technology and has not been widely adapted in software development industry. Therefore, conducting a cases study would be difficult for us to compare these GUI testing approaches.

4.2 Context Selection

It is an off-line master thesis project which was conducted by us who are two students (at their second year of master's degree in software engineering) within Blekinge Institute of Technology. Due to constraints in cost and time, this project did not address real problems faced in industry. However, we apply these two testing approaches to opensource Android projects that can be found online.

4.3 Variables

The independent variables are those that we can control and change in the experiment. The dependent variables are often used to measure the effect of the treatments. Selection of the independent variables determine the cases for which the dependent variables are sampled [22]. The variables for this research are summarized and explained in Table 4-1.

The design type of this experiment is one factor with more than two (four in particular) treatments. Though the aim of this study is to assess the performance of two testing approaches, it is not possible to using these approaches directly. Testing approaches are often implemented by automated testing tools. Hence, the treatments in this experiment are the four automated GUI testing tools we selected. Details of how we select these four tools are in section 4.6.

Table 4-1 Variables

	Name	Description
Independent variables	TOOL	Four tools are used to implement the identical test cases: UI Automator, Appium, EyeAutomate, and SikuliX.
Controlled variables	APP	The 20 open source android applications that are selected to be tested.
	CASE	The test cases that are designed to test the 20 applications.
	EXPERIENCE	The experiences of developers who are employed to implement the test cases have to be at the same level.
	PLATFORM	The platform (both hardware and software) that the test cases are being implemented and executed are controlled to be identical.
Dependent variables	IMPLEMENT	The time spent by developers to implement each test case is recorded. The time unit used is minutes.
	EXECUTE	The time spent to execute each test cases is recorded. The time unit used is seconds.
	DEFECT	The number of real defects that are find by each tool.
	FALSE	The number of false positive defects alerts when executing test cases of each tool.

4.4 Hypotheses

The objectives of the study are to compare the time spent on implementing test cases of each approach, to compare the time costed when executing test cases of each approach and to compare the number of defects found by each approach. Three null hypotheses are addressed for these objectives and to answer the three research questions we mentioned above.

$H_{0, \text{IMPLEMENT}}$. There is no difference in terms of the time spent on implementing test cases between 2nd generation and 3rd generation tools.

$H_{0, EXECUTE}$. There is no difference in terms of the time spent on executing test cases between 2nd generation and 3rd generation tools.
$H_{0, FALSE}$. There is no difference in terms of the false positive defect alerts in 2nd generation and 3rd generation tools.
$H_{0, DEFECT}$. There is no difference in terms of the real defects found by 2nd generation and 3rd generation approaches.

4.5 Subjects Selection

The experimental subjects, in this case, are the apps we choose. Because conclusions we draw will be about the apps and the GUI testing approaches. To improve the generalization of this experiment, because we stated in the aim that we would like to assess the performance of these two approaches (2nd vs 3rd generation) of automated GUI testing in terms of testing Android applications in general, one method we use is randomness. To improve the randomness of the selected apps, the mechanism we use is stated in below:

Build a pool of available open source Android applications. To build this pool, the first thing we do is to search in GitHub for open source Android projects. Besides, we selected applications according to the criteria we set. Those criteria are:

Color scheme. The color scheme is the colors of GUI elements used by the applications. This is considered because VGT uses image (screenshots) to access the GUI elements. And color is one of the major properties used in computer image recognition. VGT may perform better in the apps which has a high contrast color scheme.
Language. It is the supported language of each application. This is considered because VGT uses image (screenshots) to access the GUI elements. Text recognition is another property used in image recognition. And text can also be used by 2nd generation testing approach as the input for accessing GUI elements.
Category. The category of each application that belongs to. We have references Google Play Store for determine what category, for example, photography, music & audio, entrainment, etc., should each application should be assigned. Selecting apps from different categories would help to improve the generalization of the results.
Minimum supported Android API level. This is important because UI Automator only supports applications that are implemented in Android API level 18 or above.

Blind random selection from the pool. The second and last step is selecting twenty applications from the pool randomly. A list with brief descriptions of the selected apps are available in Table 4-2.

Table 4-2 List of selected Android apps

Name	Description	Color scheme	Language	Category	Android API level
AmazeFileManager	An open source, light and smooth file manager application with material design.	Customizable	English	Tools	27
CarbonForum	An Android client for Carbon Forum with material design.	Blue / White	English	Social	23

AntennaPod	An easy-to-use, flexible and open-source podcast manager for Android.	Customizable	English	Music & Audio	26
CalendarII	A pure calendar app in Chinese with material design.	Customizable	Chinese	Productivity	27
Currency	A simple Android currency conversion application in conventional Android design.	Customizable	English	Finance	22
FastReader	An open source e-book reader.	Customizable	Chinese	Books & Reference	23
ForkHub	An open source GitHub client with material design.	Blue / White	English	Productivity	27
GnuCash	A companion expense-tracker application for GnuCash (desktop) designed for Android with material design.	Customizable	English	Finance	27
Jockey	A music player for Android based on Google's material design standards.	Customizable	English	Music & Audio	27
LeafPic	A fluid, material-designed alternative gallery application.	Customizable	English	Photography	27
Markor	A simple and lightweight text editor supporting Markdown and todo.txt in material design.	Customizable	Customizable	Productivity	27

MovieGuide	Android app that lists popular/highest-rated movies, shows trailers and reviews.	Dark Grey	English	Entertainment	26
MoviesWorld	An app to help users discover popular and highly rated movies on the web.	Purple / White	English	Entertainment	25
NewPipe	A free lightweight YouTube frontend for Android in material design.	Red / Black / White	English	Video players	27
OmniNotes	Note taking open-source application aimed to have both a simple interface but keeping smart behavior in material design.	Blue / White	Customizable	Productivity	23
OptiMovies	An app helps users to discover various categorized Movies and TV Shows in material design.	Teal / White	English	Entertainment	25
Sanxing	A tool to help you organize your tasks, to develop a habit, and to see the time left for your life or college years.	Customizable	English	Productivity	26
SimpleCalendar	A simple calendar with events and a customizable widget.	Orange / Grey / White	English	Productivity	27

SoundRecorder	A simple sound recording app implementing material design.	Red / White	English	Music & Audio	21
Timber	A material design music player.	Customizable	English	Music & Audio	26

4.6 Testing Tools Selection

To assess the performance of the testing approaches, we are using the GUI testing tools as representatives of the two approaches. To ensure that the comparisons between these two approaches are fare, we carefully selected two tools for each approach. In this section, we discuss about what characteristics of the approach that a tool should use have and what criteria are used for selecting testing tools.

4.6.1 Tools Using 2nd Generation Approach

There are lot of 2nd generation GUI testing tools available for Android applications on the market, for example UI Automator, Espresso, Robotium, Appium, etc. We formulated some criteria for testing tools selection based on the based on previous researches [23]. The criteria we considered while selecting the tools are:

Ability of accessing GUI elements. 2nd generation approach access and locate GUI elements by different properties (id, text, class…) of the element. What type of properties that can be used by a tool to access GUI elements will be analyzed.

Logging support. To improve the simplicity of data collection (times elapsed when execution a test case and failed test cases logging) in this experiment. The logging support ability is analyzed for each tool.

Documentation. Documentation is an important aspect that may affects the learning time and the time costed for implementing test cases. Time costed for implementing test cases can be significantly reduced while the documentation is clear and detailed for testers to follow.

Script language support. Script language is another aspect that may affect the performance of testers while implementing test cases using each tool due to different testers have different proficiency of different programming languages.

Emulator support. Because there are not any tools available for Android GUI testing using 3rd generation GUI testing approach. Therefore, in order to ensure that the experiment is fare, we need the tools for supporting GUI testing on emulator.

Cross applications testing support. A GUI test that spans multiple apps lets testers verify that the app under test behaves correctly when the user flow crosses into other apps or into the system UI. For example, a file manager app calls a photo gallery app to open a photo.

Based on the criteria, we accessed five popular automated GUI testing tools, see Table 4-3. From this table, we can see that Espresso and Selendroid do not support cross applications testing. So, they are rejected. As for Robotium, because of its poor documentation of its API, it is also rejected. Finally, we selected UI Automator and Appium as the tools to represent the testing tools that using 2nd generation approach.

Table 4-3 Comparison of tools using 2nd generation approaches

Criteria	UI Automator	Appium	Espresso	Selendroid	Robotium
GUI elements accessibility	Good	Good	Good	Good	Good
Logging support.	Good	Good	Good	Good	Good
Documentation	Good	Good	Good	Good	Poor
Script language support	Java	Java, Python	Java	Java, Ruby	Java
Emulator support	Yes	Yes	Yes	Yes	Yes
Cross applications testing support	Yes	Yes	No	No	Yes

4.6.2 Tools for 3rd Generation Approach

For 3rd generation testing tools, our choice is very limited. Because, it is still a new technology for GUI testing of Android applications. After searching on the Internet, we only found two visual GUI testing tools, EyeAutomate and SikuliX. EyeAutomate uses a combination of image recognition and artificial intelligence for VGT. It works on all apps (desktop, web and mobile). SikuliX is an open source tool that uses image recognition technology to conduct GUI testing. The using of SikuliX on native Android device is in an early experimental stage. Though these two tools both claim that they support VGT for Android application. Currently, they only support VGT for Android applications on Android emulators in desktop environment.

4.7 Test Case Design

To ensure that the comparisons between tools are fair, we first define a generic set of test cases using natural language, and then implement the exact same set of test case in all the tools that we are comparing. Through this way, all the tools are evaluated based on the same task. Test cases for each Android application with brief description are listed in Table 4-3.

Table 4-4 Test case design

Test Case No.	App Name	Description
1.1	AmazeFileManager	Create new folder
1.2		Create new file
1.3		Change theme
1.4		Change color
1.5		Open file
1.6		Add to bookmark
2.1	CarbonForum	Open a topic
2.2		Reply to a topic
2.3		Refresh
2.4		See notification
3.1	AntennaPod	Add Podcast
3.2		Add a episode to queue
3.3		Stream an episode
3.4		Download, delete an episode
3.5		Clear queue
3.6		Change theme
3.7		Check download status
4.1	CalendarII	Add events
4.2		Delete selected event
4.3		Delete all event
4.4		Jump to a specific date
4.5		Back to today
4.6		Change theme
5.1	Currency	Add currency
5.2		Input amount
5.3		Edit currency
5.4		See exchange rate chart
5.5		Change fraction digits
5.6		Change theme
5.7		See help
5.8		Refresh

Test Case No.	App Name	Description
6.1	FastReader	Add book
6.2		Open book
6.3		Change font size
6.4		Change background color
6.5		Delete a book
6.6		Change to dark mode
7.1	ForkHub	Search
7.2		See my repositories
7.3		Create a gist
7.4		See notifications
7.5		Log out
8.1	GnuCash	Add a book
8.2		Rename a book
8.3		Add an account
8.4		Add a transaction
8.5		Edit account
8.6		Delete account
8.7		See reports
8.8		Search
9.1	Jockey	Play, pause a track
9.2		Next track
9.3		Shuffle all
9.4		Enable Equalizer
9.5		Create new playlist
9.6		Add to playlist
9.7		Create new smart playlist
9.8		Change color
10.1	LeafPic	Open a picture
10.2		Rotate
10.3		Delete a picture
10.4		Change theme
10.5		Show palette
10.6		Show data
10.7		See about
11.1	Markor	Create notes
11.2		Create Folder
11.3		Open setting
11.4		Change language
11.5		Create To-Do
11.6		Create Quick Note
11.7		Delete notes

Test Case No.	App Name	Description
12.1	MovieGuide	Skim movies
12.2		See movie details
12.3		Add to favorites
12.4		See trailers
12.5		Change sorting
13.1	MoviesWorld	Skim movies
13.2		See movie details
13.3		Add to favorites
13.4		See trailers
13.5		Change sorting
14.1	NewPipe	Skim video list
14.2		See subscriptions
14.3		Play video
14.4		Search
14.5		Add to play list
14.6		Change Theme
15.1	OmniNotes	Create a photo note
15.2		Create a checklist
15.3		Create a text note
15.4		Archive notes
15.5		Merge notes
15.6		Change language
15.7		Change sorting
16.1	OptiMovies	Skim movies
16.2		See movie details
16.3		Add to bookmarks
16.4		Skim tv shows
16.5		Change region
16.6		See about
17.1	Sanxing	Create a new task
17.2		Create a new habit
17.3		Create a new countdown
17.4		See calendar
17.5		See statics
17.6		See timeline
18.1	SimpleCalendar	Add events
18.2		Edit event
18.3		Change view
18.4		Back to today
18.5		Change theme

Test Case No.	App Name	Description
19.1	SoundRecorder	Start recording
19.2		Play records
19.3		Rename file
19.4		Delete file
20.1	Timber	Play, pause a track
20.2		Create new playlist
20.3		Add to playlist
20.4		Change theme

4.8 Defects Injection

To make it easy to find defects, we manually seed 10 defects to the applications. Due to out limited knowledge of Android development, the mechanism we used of seeding defects is through removing or blocking listener of a GUI element without removing the GUI elements from the interface. Through this, when an element is clicked, the listener would not perform as usual and a defect will appear.

Table 4-5 Seeded defects

Defect No.	Related Test Case No.	App Name	Description
1	2.4	CarbonForum	See notification
2	4.4	CalendarII	Jump to a specific date
3	4.5	CalendarII	Back to today
4	5.7	Currency	See help
5	7.4	ForkHub	See notification
6	9.7	Jockey	Create new smart playlist
7	10.7	LeafPic	See about
8	13.4	MoviesWorld	See trailers
9	16.6	OptiMovies	See about
10	17.6	Sanxing	See timeline

These defects were seeded because of the simplicity of understanding and modifying the source code of the selected application.

5 EXPERIMENT OPERATION

This section presents the process of how this experiment is prepared and executed.

5.1 Preparation

The first thing we did for preparation was to set up the runtime environment for the experiment. The PC we use to implement and execute the test cases has the following specifications:

Table 5-1 Specifications of the environment

Memory	16 GB
Processor	AMD Ryzen 5 2600X Six-Core Processor @ 3.8 GHz
Graphics	GeForce GTX 1050 Ti
Disk	240 GB SSD
OS ver.	Ubuntu 16.04 LTS 64-bit
Android Studio	Android Studio 3.0.1
UI Automator	UI Automator v18:2.1.3
Appium	Appium v1.9.0 Appium java client v6.1.0
EyeAutomate	EyeAutomate v2.1
SikuliX	SikuliX IDE v1.1.2

Before conducting the experiment, we spent several weeks to build our knowledge on Android platform and Android application development. Afterwards, we spent time on learning to use those 4 testing tools. An exercise session was held by us, where we used those 4 testing tools to implemented test cases for a Calendar app.

To ensure randomization and improve validity, we randomly selected 20 opensource Android applications. Those applications are in different categories, different color scheme, different language, different API level, etc., see Table 4-2. After all the applications are founded. We manually seed 10 defects to the applications, i.e., we removed or blocked some of the listeners without removing the GUI elements. Through this, the GUI is tested, but the listener would not perform as usual and a defect will appear.

To ensure that the comparisons between tools are fair, ahead of conducting the experiment, we designed a generic set of test cases (120 test cases for 20 Application) using natural language. All the test cases are listed in Table 4-4. And when we were implementing the test cases, we used it as a guide. Through this way, all the tools were evaluated based on the same task.

5.2 Execution

When implementing the test cases, we need to ensure that all the teat cases are implemented by testers whose programming abilities and experiences are at the same level, so the time needed for implementing test cases are comparable and fair. To ensure this, the approach we use is paired programming. Because

we only have two developers available. Two roles were set to the two developers. One was the driver, and the other was the observer. Driver wrote codes, and the observer reviewed each line of code as it was typed in. We implemented all the test cases in a sequence order against the number we give to each test case. As for the sequence of tools we implemented the test cases with, we first implemented the test using UI Automator, then Appium, followed by EyeAutomate, finally SikuliX. The programming language we used for UI Automator and Appium was Java. As for EyeAutomate and SikuliX, we used visual language provided by their IDEs.

The observer also took the responsibility of recording the time of implementation for each test case. An Excel sheet was also generated for the use of recording the time. The implementation time we record here is the time that are really spent on development the code. The time we spent on debugging and adjusting the code (which usually took much longer than it was spent on typing in code) was not in our consideration. We have rejected to including the debugging time, because time spent on dubbing is more depend on the ability of the testers and the code they write. While the difference of tools only small and limited effects on the debugging time. In this experiment, we focus on the differences between tools rather than testers.

After all the test cases were implemented, we executed all the test case one by one. At the meantime, the time took for each test case to execute, and the status (passed or failed) of each test case is recorded. For each failed test case, we reviewed it to check if it was a false positive or a real defect. As for real defects, we shall not just consider the defects we seeded before implementing test cases, but also there may be other real defects existing in the selected apps. Because, most of the open source apps are developed by a small group of developers and the quality of the applications are not assured.

6 Data Analysis and Interpretation

This section presents the statistical analysis of the data that we gathered from implementing and executing test cases. Data are analyzed to compare with the hypotheses we made and to try to answer the research questions that we stated above.

6.1 Performance in Terms of Implementing Test Cases

To compare the performance of each GUI testing tool in terms of implementing each test cases, the time we spent on implementing each test case was recorded. Box-plot of time spent on implementing each test case using each generation approach (IMPLEMENT) is shown in Figure 6-1. For IMPLEMENT, 2nd generation approach has higher mean that 3rd generation approach. While the minimum of 2nd generation approach is slightly higher than 3rd generation approach. However, the maximum of 2nd generation approach is much greater than the maximum of 3rd generation approach.

Besides, there are few outliers in each generation approach. These outliers are existing because sometimes we quiet a lot of time on looking up through the guidelines of each tool to find the usage of a function.

Besides the box-plot diagram, we also performed a statistical significance analysis using p-value. P-value is used here to assess if the difference is the result of a difference between methods, or just randomness in the data. The significance threshold was set at 0.05. The p-value of this two groups of data is less than 0.001, which means this difference is not just randomness.

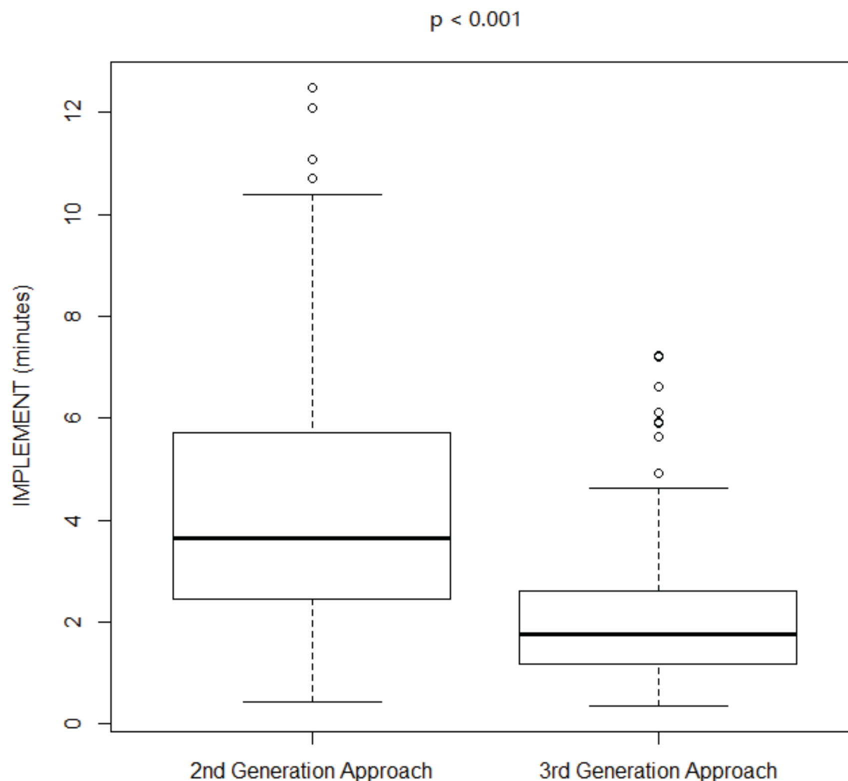

Figure 6-1 Box plot for IMPLEMENT

Beside analyzing the average time spent on implementing each test case, we also make a comparison of the total time spent on implementing test cases of each application, as shown in Figure 6-2. From the bar chart, we can see that in nineteen out of twenty applications, UI Automator and Appium take longer time to implement test cases than EyeAutomate and SikuliX. While there is only one exception that in implementing test cases for Markor, Appium take less time than EyeAutomate and SikuliX.

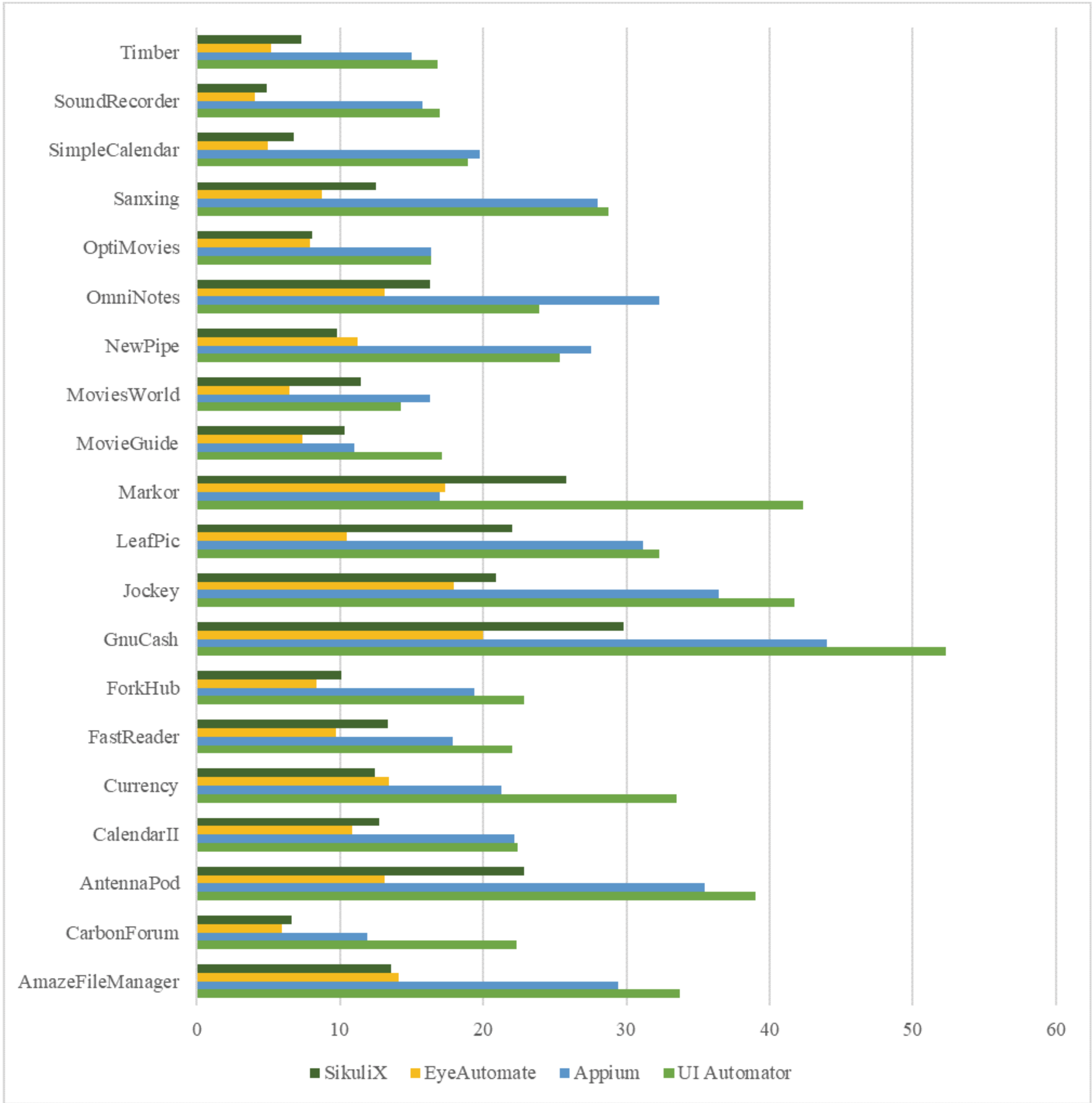

Figure 6-2 Bar chart of total IMPLEMENT for each APP (in minutes)

6.2 Performance in Terms of Executing Test Cases

To compare the performance of each GUI testing tool in terms of executing test cases, the time that the system took to execute each test case was recorded. For execution time, as we can see in this boxplot figure, 2nd generation approach has a higher median than third generation approach. In terms of the minimum, 2nd generation approach is also slightly higher. While the maximum of 2nd approach is much higher than the maximum of 3rd generation approach.

Besides, there are quite a few outliers in both approaches. The outliers were the results of failed test cases. When a test case is failed to run, it usually took a very long time for it to terminate.

Statistical significance is also calculated here using p-value. The significance threshold was set at 0.05. The p-value of this two groups of data is less than 0.035, which means this difference is not just randomness.

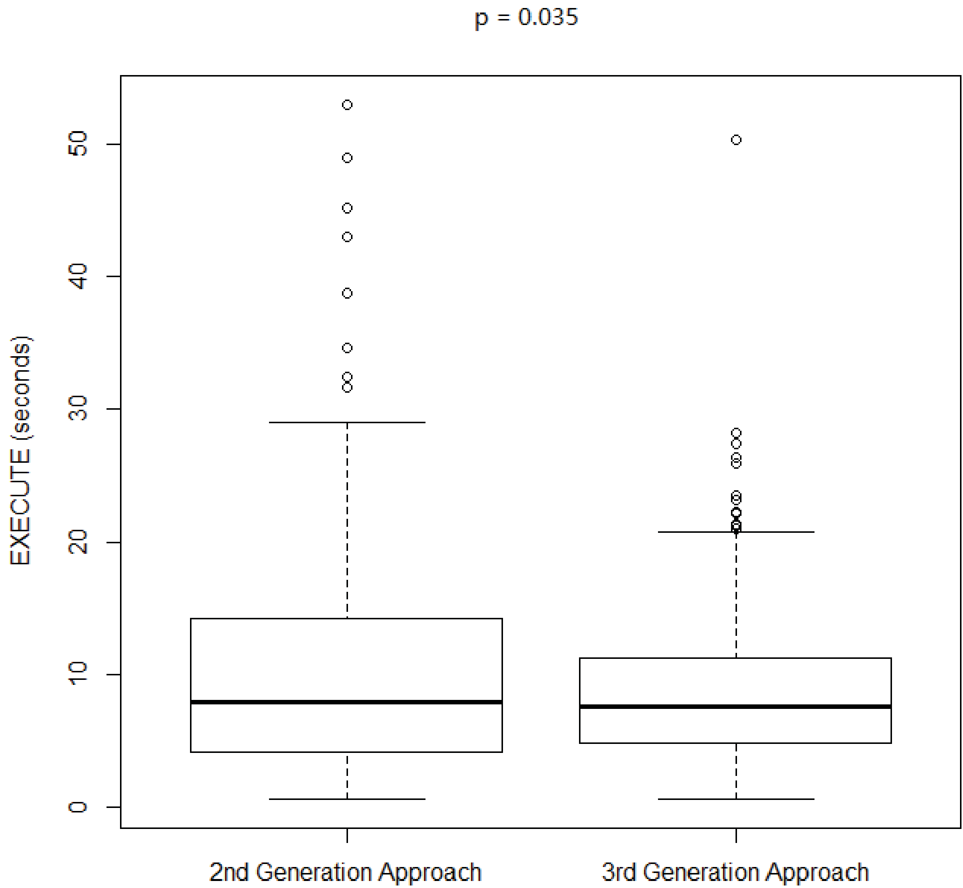

Figure 6-3 Box plot for EXECUTE

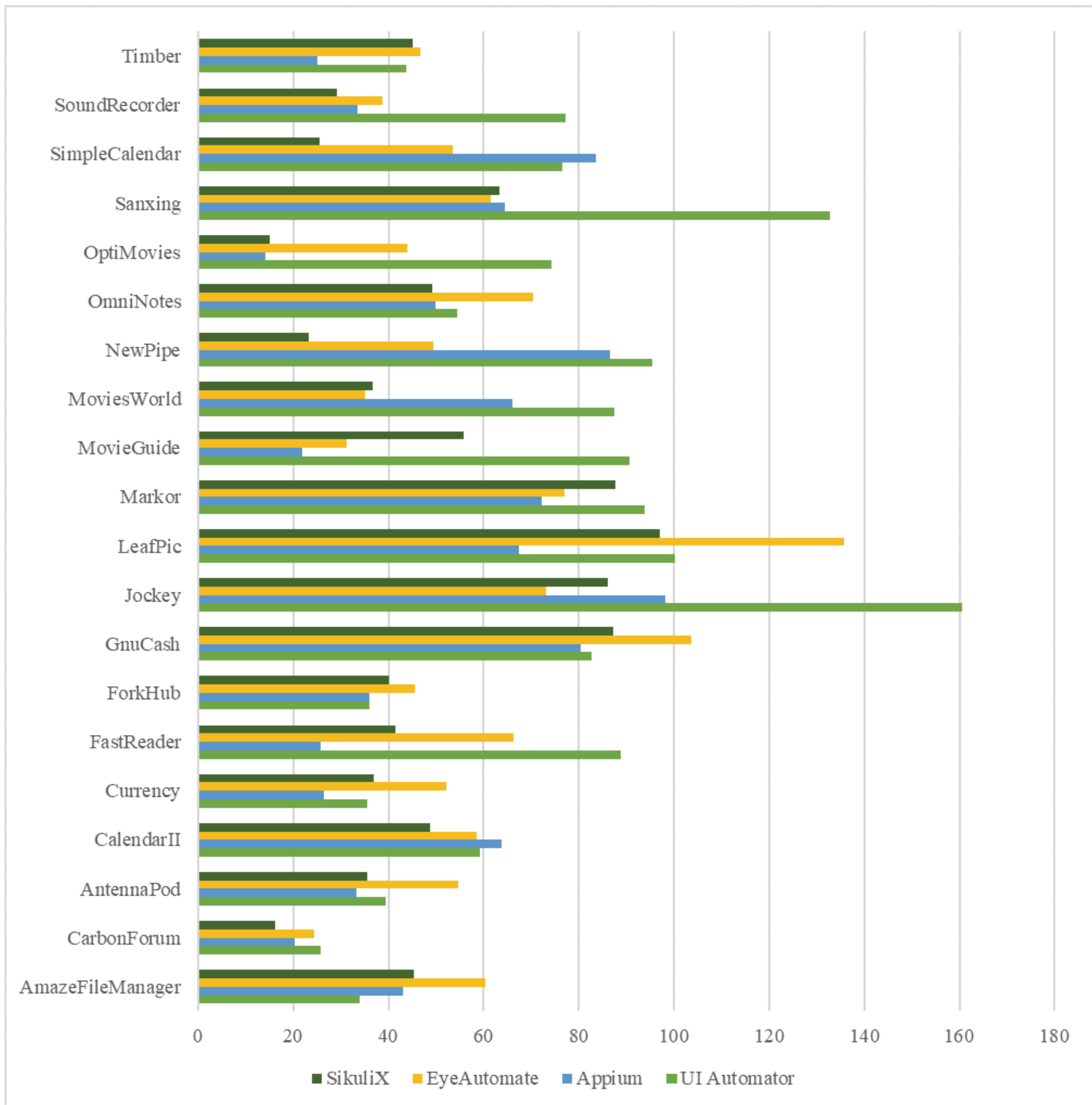

Figure 6-4 Bar chart of total EXECUTE for each APP (in seconds)

Beside analyzing the average time spent on implementing each test case, we also make a comparison of the total time spent on executing test cases of each application, as shown in Figure 6-4. It is different for each application and each tool. In some applications, tools using 2nd generation tools took longer time to execute test cases, this is happening because in those applications we implemented test cases that need to type in text, type in text using 2nd generation approach took long time. While in others, 3rd generations tools took longer time to execute test cases, this is happening often because in those applications the GUI is often complicated (i.e., the GUI uses many colors, complicate design), image recognition in this environment took long time.

6.3 Defects Found by Different Tools

In terms of the seeded defects, all the tools found all the defects that we seeded before. This happened, because the developers know all the defects before designing the test cases. And then the test cases were designed intend to find the defects. Also, because the defects we seed is simply removing listeners of the GUI button, would be too easy for the testing tools to detect these types of defects.

Besides, the defects we seed, the tools also found other defects that may be true positive, but after we investigated into the logs and run the test cases manually, we found those defects occurs randomly. And after switching device or emulator, those defects never happen again. Therefore, in this data analysis, we regard those defects as false positive.

A bar chart is drawn in terms of the false positives that appears while using each testing tools. Bar chart is shown in Figure 6-5. False positives appeared in a much higher frequency in EyeAutomate and SikuliX than UI Automator and Appium.

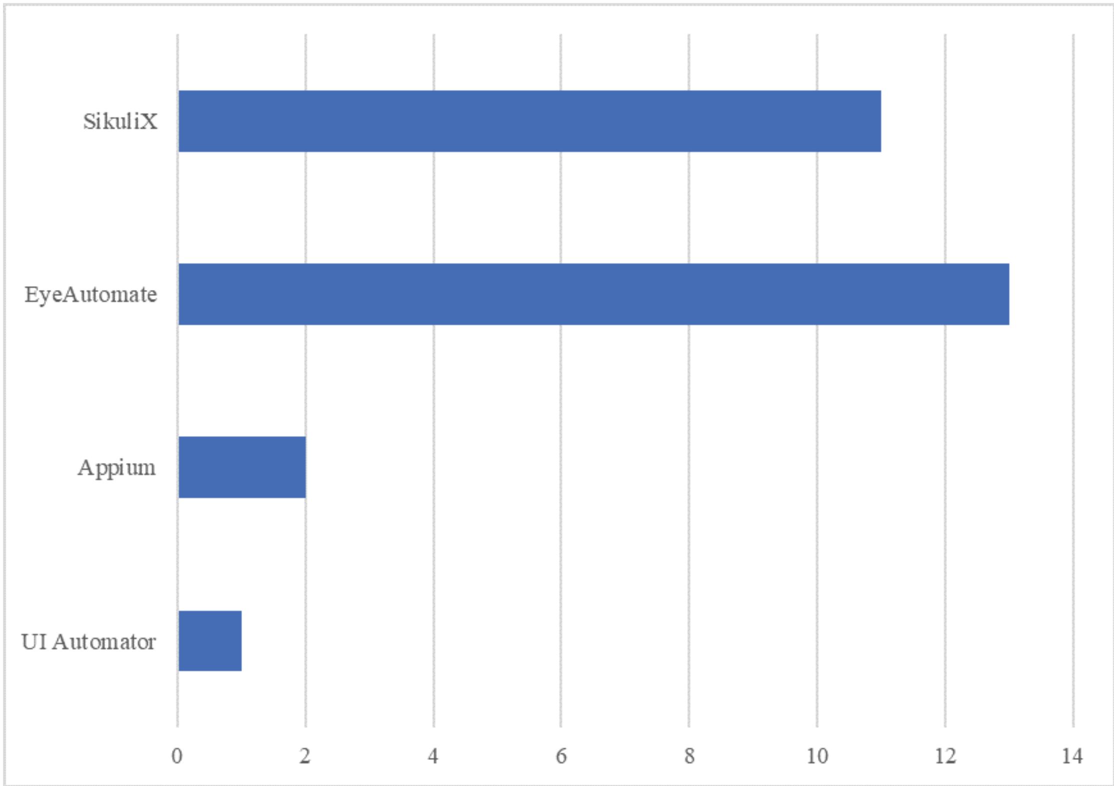

Figure 6-5 Bar chart of false positives

By analyzing the false positives founded by each tool, we found that tools using 3rd generation approach find more false positives, because it is not robust to color scheme change and language change. When the color scheme and the language changes, the appearance of the GUI is changed. And it will be difficult for computer to recognize those changes, and to find the matching image (screenshot).

6.4 Results Interpretation

From the research above and the hypotheses we formulated in Section 4.3, we can interpret the results as follows:

$H_{0,\ IMPLEMENT}$ can be rejected. In terms of the time spent on implementing test cases, 2nd generation approach takes much more time than 3rd generation approach.

$H_{0,\ EXECUTE}$ can be rejected. The p-value we get is 0.035, which is less than the threshold. But there is no clear pattern of the comparison in terms of the time spent on executing test cases between 2nd generation and 3rd generation tools. It is quiet depend on the design of the test case, and the applications.

$H_{0,\ FALSE}$ can be rejected. 2nd generation tools find much less false positives than 3rd generation tools.

$H_{0,\ DEFECT}$ is accepted. Because all the tools found the exact same defects. In this particular experiment, there is no difference in terms of the real defects found by 2nd generation and 3rd generation approaches.

7 THREATS TO VALIDITY

The validity of the results is a fundamental question concerning results from an experiment. There are four types of threat to the validity of the results, conclusion validity, internal validity, construct validity and external validity [22]. Below, we explain the major threats to validity existed in this research.

Selection of testers were not randomized. Due to the limited resources we have of testers available, we only used two testers to implement the test cases. However, we conducted paired programming while implementing test cases. Therefore, all the test cases for all the tools are implemented by the same pair of programmers. This helped us to control that the experiences of testers are at the same level. Therefore, the results we collected are still comparable. While, results might differ, if other testers repeated this experiment. Thus, this is a threat to validity.

Selection of Android apps were not randomized. In this specific experiment context, only those small-scale open source Android applications, both in terms of the size of development groups and line of code of applications, were tested. Thus, selection of Android apps might not be randomized enough. Results might differ, if other people repeated this experiment on different Android applications.

Design of test cases are limited. The test cases were only designed to test if certain functions were fulfilled by those applications. But in a real Android development project, there would be much more different types of test case designs, e.g., testing latency, testing stability, etc. Therefore, the test cases we used in this experiment would not represent all the types of test cases used in industrial Android development projects.

Learning effects existed and were not mitigated. We did not randomize the sequence of using each tool for test case implementation. Instead, we used UI Automator to implement all the test cases first. And then used Appium to implement all the test cases again. After that, we used SikuliX to implement all the test cases for the third time. Lastly, all the test cases were implemented using EyeAutomate again. Therefore, implementation time may be affected by maturation. This might affect the result hugely.

Only one type of defects was tested. We only tested only one type of defects in this experiment, i.e., non-working listener. And this might have resulted that all the injected defects we detected by all the tools. And we did not find differences of different testing approaches in terms of detecting real defects. However, these two approaches might differ in terms of detecting other types of defects, for example, defects like GUI elements that are not rendered on the screen but existed in the source code.

3rd generation tools were not dedicated to Android testing. Because 3rd generation tools used in this experiment were not dedicated to Android testing, therefore, the results might affect by that. But 3rd generation tools are designed to test the application from a user's perspective of sensing the applications under test using computer vision. This approach was not bounded by the platform which the applications under test were running on.

8 CONCLUSION

In this section, we present the lessons learned and conclude this study.

8.1 Lessons learned

Through this study, we have learned a lot about GUI tastings on Android applications. Meanwhile, we have also investigated many issues we did not notice before. Below, we present the lessons we learned from this study.

The time need for learning to use 2nd generation GUI testing approach is high. Because the 2nd generation relies on tags or IDs of GUI components and widgets to accessed GUI elements. It requires testers to have some knowledge of source code of the applications that are being tested. And in Android GUI testing, testers may should also have knowledge in using Android SKDs and the tools provided by it to access Android Activity name, package name and other properties of GUI components, in order to test the application. Besides, it requires testers to have knowledge of programming languages, because, the testing tools using this approach often use Java, JavaScript, Python, etc. as the language to implement test cases. Therefore, this approach is not so friendly to people who are not familiar with Android development. Thus, for beginners in Android GUI testing, the learning cost of using this approach is relatively high.

2nd generation GUI testing approach has difficulties in testing dynamic GUI components. While using 2nd generation approach to test a fragment of GUI that includes dynamic components such as progress bar within a music or video player, it will need quiet a long time to locate a GUI element or even fail to find the element frequently. Besides, while a fragment of GUI that includes dynamic components, using UI Automatorviewer (a tool provided by Android SDK for access the properties of GUI components) to obtain the GUI components will be impossible in some occasions.

Implementing typing from onscreen keyboard using 2nd generation GUI testing approach is difficult. When using UI Automator or Appium to implement a test cases that needs to type in some word from the onscreen virtual keyboard of the phone, it requires quiet a lot manual labor. Because the approaches provide by these tools are implemented in a quiet complicated way. Even after it is implemented, to execute the actions of typing on onscreen keyboard also requires those tools a quiet long time.

2nd generation GUI testing approach differs quiet a lot in each version. In UI Automator and Appium, the difference between each version is quite big. Especially in Appium, many functions are dumped or are completely rewritten in other ways. Therefore, maintaining test cases in order to adapt to different version is quite difficult, and quite costly.

3rd generation GUI testing approach is not robust to screen size changing. While using 3rd generation testing tools on different platforms, especially when the resolution of the screen is different, the test cases developed on one machine could not be applied to another. Even on the same machine, if the window of the Android emulator was resized, the test case will fail.

3rd generation GUI testing approach is not robust to color change of the application. In Android platform, many applications support to customization, such as change color and change theme. But test cases implemented using3rd generation GUI testing approach cannot adapt those changes. Because this testing approach uses computer vision (image recognition) to locate GUI elements. And color is one of the critical criteria of image recognition. After the color is changed, it will be very difficult for those tools to use the original image to match the GUI components on the Android emulator. In this experiment, we designed some test cases to test the function of changing color scheme of some application. We found that after we changed the color scheme of the application, test cases implemented before changing color scheme will not work afterwards. Therefore, the image recognition techniques used by these two tools (SikuliX and EyeAutomate) are color sensitive and not robust to color changes.

3rd generation GUI testing approach is inefficient in testing complex GUI frames. When a frame of GUI is complicate, i.e., the frame of GUI has a lot of different elements, has multiple pictures, or has many similar GUI elements, it will be difficult for the tools to locate the element properly and sufficiently, or it will take a very long time for the tools to find the correct GUI element. Abnormal behaviors will appear while executing those test cases.

3rd generation GUI testing approach is inefficient in testing Applications that changes frequently. Some the applications will frequently change its layout due to the content of those application changes frequently, e.g., YouTube player, podcast player, forum client, etc. While the content of the applications is changed, the test cases developed before the change will not be applied to the new content.

3rd generation GUI testing approach does not support native Android device. Currently, there is not tools that supports to apply 3rd generation GUI testing approach to native Android device. There are some experimental features provided by SikuliX to run 3rd generation GUI testing on Android device. But implement a customized test case is still impossible. And because of this, some gestures that are designed to run on a mobile device could ne be tested using these tools.

8.2 Conclusion

For **RQ1**, tools using 3rd generation GUI testing approach take less time to implement test cases than tools using 2nd generation GUI testing approach. For **RQ2**, there is not a specific pattern when comparing tools using 2nd and 3rd generation GUI testing approaches in terms of time cost on executing test cases. It is different between different test cases. For **RQ3.1**, false positives appear much more often in tools using 3rd generation GUI testing approach than tools using 2nd generation GUI testing approach. For **RQ3.1**, real defects found by each tool are the same.

To conclude, 3rd generation GUI testing approach is more efficient in terms of implementing test cases than 2nd generation GUI testing approach. But 3rd generation GUI testing approach finds much more false positives than 2nd generation approach. To decide if a defect alert is false positive or not requires human effort. In a long term, it may accumulate huge lost on human efforts. To maintain test cases, 3rd generation approach consumes lots of human efforts.

9 FUTURE WORK

In this section, we make some suggestions of what to do in the future.

As we mentioned in Section 7, implementation time may be affected by maturation. To mitigate this validity threat, if we have chance to conduct this research again, we could randomize the sequence for implementing test cases. We could use implement test cases one by one using all the tools and randomize the sequence of each testing tool for each test case. Through this, we could reduce the impact of maturation ad improve the validity.

Another thing we could do is to investigate the performance of these two testing approaches on other types of defects. As we mentioned in section 7, we only tested one type of defects. But the performance of the tools might differ on other type of defects based on their stipulated capabilities. For example, if a GUI element were rendered out of the screen, 2nd generation tools might still trigger the action of this GUI element and could not detect this defect. But 3rd generation tools might detect such type of defects because it could not find the GUI element based on the image appeared on the screen.

One thing we can do in the future is to investigate the feasibility of applying 3rd generation GUI testing approach on native Android device. The is already some experiment features shown in SikuliX for supporting VGT on native android device.

REFERENCES

[1] M. A. Mohd Shukran and W. S. S. B. Sharif, "Android augmented reality system in Malaysia military operations - Unit positions," vol. 6, pp. 79–82, Aug. 2012.

[2] "Mobile OS market share 2018," *Statista*. [Online]. Available: https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/. [Accessed: 13-Oct-2018].

[3] N. Gandhewar and R. Sheikh, "Google Android: An Emerging Software Platform For Mobile Devices," *Int J Comput Sci Eng*, vol. 1, 2009.

[4] "How to install the Android SDK (Software Development Kit)." [Online]. Available: https://www.androidauthority.com/how-to-install-android-sdk-software-development-kit-21137/. [Accessed: 14-Oct-2018].

[5] "App stores: number of apps in leading app stores 2018," *Statista*. [Online]. Available: https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/. [Accessed: 14-Oct-2018].

[6] I. Banerjee, B. Nguyen, V. Garousi, and A. Memon, "Graphical user interface (GUI) testing: Systematic mapping and repository," *Inf. Softw. Technol.*, vol. 55, no. 10, pp. 1679–1694, Oct. 2013.

[7] J. Srivastaval and T. Dwivedi, "Software Testing Strategy Approach on Source Code Applying Conditional Coverage Method," 2015.

[8] P. Aho, N. Menz, T. Räty, and I. Schieferdecker, "Automated Java GUI Modeling for Model-Based Testing Purposes," in *2011 Eighth International Conference on Information Technology: New Generations*, 2011, pp. 268–273.

[9] Y. Baek and D. Bae, "Automated model-based Android GUI testing using multi-level GUI comparison criteria," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, pp. 238–249.

[10] E. Alégroth and R. Feldt, "On the long-term use of visual gui testing in industrial practice: a case study," *Empir. Softw. Eng.*, vol. 22, no. 6, pp. 2937–2971, Dec. 2017.

[11] E. Alégroth, R. Feldt, and P. Kolström, "Maintenance of automated test suites in industry: An empirical study on Visual GUI Testing," *Inf. Softw. Technol.*, vol. 73, pp. 66–80, May 2016.

[12] E. Alégroth, M. Steiner, and A. Martini, "Exploring the Presence of Technical Debt in Industrial GUI-Based Testware: A Case Study," in *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2016, pp. 257–262.

[13] E. Alegroth, Z. Gao, R. Oliveira, and A. Memon, "Conceptualization and Evaluation of Component-Based Testing Unified with Visual GUI Testing: An Empirical Study," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, 2015, pp. 1–10.

[14] E. Alégroth, *Visual GUI Testing: Automating High-level Software Testing in Industrial Practice*. 2015.

[15] Y. Chen and H. Jiang, "Comparison of Different Techniques of Web GUI-based Testing with the Representative Tools Selenium and EyeSel," Blekinge Institute of technology, Karlskrona, Sweden, 2017.

[16] M. Grechanik, Q. Xie, and C. Fu, "Creating GUI Testing Tools Using Accessibility Technologies," in *2009 International Conference on Software Testing, Verification, and Validation Workshops*, 2009, pp. 243–250.

[17] V. Garousi *et al.*, "Comparing Automated Visual GUI Testing Tools: An Industrial Case Study," in *Proceedings of the 8th ACM SIGSOFT International Workshop on Automated Software Testing*, New York, NY, USA, 2017, pp. 21–28.

[18] R. Potter, in *Triggers: Guiding automation with pixels to achieve data access*, University of Maryland, Center for Automation Research, Human/Computer Interaction Laboratory, 1992, pp. 361–382.

[19] E. Borjesson and R. Feldt, "Automated System Testing Using Visual GUI Testing Tools: A Comparative Study in Industry," in *Verification and Validation 2012 IEEE Fifth International Conference on Software Testing*, 2012, pp. 350–359.

[20] P. Siponen, "Test Automation via Graphical User Interface," UNIVERSITY OF VAASA, Helsinki, Finland, 2016.

[21] A. Ruiz and Y. W. Price, "GUI Testing Made Easy," in *Testing: Academic Industrial Conference - Practice and Research Techniques (taic part 2008)*, 2008, pp. 99–103.

[22] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Berlin: Springer, 2012.

[23] Meiliana, I. Septian, R. S. Alianto, and Daniel, "Comparison Analysis of Android GUI Testing Frameworks by Using an Experimental Study," *Procedia Comput. Sci.*, vol. 135, pp. 736–748, Jan. 2018.

Faculty of Computing, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden