Faculty of Engineering and Technology

Master of Software Engineering (SWEN)

Thesis

Android App Testing: A Model for Generating Automated Lifecycle Tests

اختبار تطبيقات الأندرويد: نموذج للتوليد الآلي لاختبارات دورة الحياة

'This thesis was submitted in partial fulfillment of the requirements for the Master's Degree in software engineering from the Faculty of Graduate Studies, at Birzeit University, Palestine'

By

Student Name: Malik Motan

Student Number: 1165317

Supervised By: Dr. Samer Zein

جامعة بيرزيت
BIRZEIT UNIVERSITY

Android App Testing: A Model for Generating Automated Lifecycle Tests

---

Author

Malik Motan

This thesis was prepared under the supervision of Dr. Samer Zein and has been approved by all members of the examination committee.

Dr. Samer Zein, Birzeit University
(Chairman of the committee)

Dr. Ahmad Tamrawi, Birzeit University
(Member)

Dr. Abdel Salam Sayyad, Birzeit University
(Member)

Date of Defense

August 29th, 2020

Abstract

Android is currently the dominating OS in the market. An immense number of Android apps is deployed to the Google Play store every year. Android apps are no longer merely focusing on entertainment or socialization. In fact, the literature shows that apps specializing in critical domains such as health, education and even the military are growing in numbers. This puts more pressure on app developers to produce quality apps. Research shows that current Android app testing approaches rely heavily on manual testing. Research in automatic test generation for Android apps focuses mostly on automated GUI testing, with some approaches introducing model-based testing for test case inputs. However, no studies focus on generating lifecycle tests automatically, especially for testing lifecycle method conformance. In this research, we present a model-based solution approach as a tool to conduct assertion lifecycle tests automatically for Android activity lifecycle callback methods. Our objective is to build a framework to generate such a model. Finally, we evaluated our proposed framework in two ways. First using a group case study. Then we evaluate our work using 10 real world open source Android applications. The results of our evaluation are promising and show that our proposed framework is useful for detecting errors.

المستخلص

نظام تشغيل أندرويد حاليًا هو نظام التشغيل المهيمن في السوق. يُنشر عدد هائل من التطبيقات في متجر جوجل بلاي في كل عام. لم تعد تطبيقات أندرويد تركز فقط على الترفيه أو التواصل الاجتماعي. في الواقع ، تُظهر الأبحاث أن التطبيقات المتخصصة في المجالات الهامة مثل الصحة والتعليم وحتى في الجيش في تزايد مستمر. يضع هذا مزيدًا من الضغط على مطوري التطبيقات لإنتاج تطبيقات عالية الجودة. تظهر الأبحاث أن أساليب اختبار تطبيقات أندرويد الحالية تعتمد بشكل كبير على الاختبار اليدوي. تتركز الأبحاث في مجال الاختبارات الأوتوماتيكية لتطبيقات أندرويد في الغالب على الاختبار الآلي لواجهة المستخدم ، مع بعض الأساليب التي تقدم الاختبار المستند إلى نموذج لمدخلات حالة الاختبار. ومع ذلك ، لا توجد دراسات تركز على توليد اختبارات دورة الحياة تلقائيًا, خاصة لاختبار مطابقة دورة حياة طرق نظام الأندرويد. في هذا البحث، نقدم نهج الحل القائم على النموذج كأداة لإجراء اختبارات دورة الحياة للتحقق تلقائيًا لطرق دورة حياة الأندرويد. هدفنا هو بناء إطار عمل لإنشاء مثل هذا النموذج. أخيرًا ، قمنا بتقييم إطار العمل المقترح بطريقتين. أولاً باستخدام دراسة جماعية. ثم بعدها قمنا بتقييم عملنا باستخدام 10 تطبيقات أندرويد حقيقية مفتوحة المصدر.تظهر التقييمات أن نتائج عملنا واعدة, وتوضح أن إطار العمل المقترح مفيد لاكتشاف الأخطاء.

Table of Content

List of Figures

List of Tables

List of Acronyms

IEEEXplore: Institute of Electrical and Electronics Engineers Explore

ACM: Association for Computing Machinery

AST: Automation of Software Test

ICITCS: International Conference on IT Convergence and Security

ICSRS: International Conference on System Reliability and Science

ICT4M: International Conference on Information and Communication Technology for The Muslim World

GUI: Graphical User Interface

API: Application Programming Interface

ESG: Event Sequence Graph

RNN: Recurrent Neural Network

AST: Abstract Syntax Tree

AUT: Application Under Test

Chapter 1

Introduction

This chapter introduces research aim, objectives, and motivation.

1.1 Introduction and Motivation

Nowadays, rapidly evolving mobile apps are revolutionizing the way people live and interact in all aspects of modern life. Mobile apps are ubiquitous and cover a wide spectrum of domains. Technology startups are reshaping the way people live by presenting creative and highly intelligent mobile apps. In fact, these apps are no longer merely targeting entertainment and socialization, they're now being applied in more critical domains such as health, education and even military [1], [2] to mention just a few. Furthermore, e-payments and m-governments mobile apps have been among the most critical types of apps these days.

The need for high quality mobile apps has never been higher. Hence, in order to make sure a mobile app functions correctly and its data integrity is preserved and not lost in the operating system, mobile apps need to conform to a standard application lifecycle model. A mobile app lifecycle model normally means the state of the app's process and whether it's paused, running, stopped, ..etc as well as the state changes from one state to another. In general, a mobile app conforms to the application lifecycle when it is implemented correctly by interacting with and transitioning properly between the application lifecycle states [3]. In order to make sure an app conforms to the application lifecycle, proper testing and validation need to be conducted.

Testing mobile apps is anything but straightforward. This is due to the diverse nature of mobile apps and platforms. Such diversity exists due to several factors including the wide range of current mobile screen sizes starting from small smartphones and all the way to tablets, the

variety of input mechanisms like keyboard, gesture, finger and even voice, the storage capacity, memory, bandwidth and processor. Consequently, it becomes difficult to test the mobile app and ensure it functions efficiently in all circumstances.

Several techniques are used to test mobile applications. Many researchers have explored using these techniques on mobile applications [4, 5, 6, 7, 8]. These techniques cover the categories of performance testing, unit testing, usability testing and functional user interface testing. However, little studies focus on test case generation for application lifecycle. Hence, the focus in this thesis research.

We chose Android as our target platform for this research for several reasons. First and foremost, Android platform has the largest market share of mobile devices of 87% as of 2019 [9]. Also, Android is an open source platform, which allows for more research freedom when exploring the platform.

1.2 Research Objectives and Problem Statement

Android app testing is an active area of research. Even though several studies have explored generating test cases for Android applications in various areas, mainly user interface, research for generating test cases to assess the quality of Android lifecycle code is still nowhere to be found. Testing the quality of the app lifecycle code is a major concern especially when producing high-end Android apps.

Objectives

In this research, we aim to:

1. Investigate how to generate a model to represent the lifecycle callback methods from the perspective of system resources.

2. Design a framework to extract such a model using static code analysis techniques.

3. Extend the framework to generate and execute automated tests from extracted model.

1.3 Structure of This Report

This report starts with a quick introduction about the research motivation, aim and objectives. Then we proceed to chapter 2 to discuss the background by explaining what Android activity lifecycle model is and how it is used in the Android app development process.

In chapter 3, we introduce the literature review. In this chapter, we present an overview of the current state-of-the-art of Android application testing and automated test generation approaches. We mainly focus on model-based test generation research for Android application and finally highlight the research gap we're trying to fill. Finally, in chapter 4 we briefly introduce our research methodology. We go over the initially suggested model of lifecycle test generation for Android activity lifecycle. After that, we conclude this report with our references.

Chapter 2

Background

2.1 Android Activity Lifecycle Model

Android applications go through a series of state changes as the user navigates through these apps. An application state can be running, paused or even closed. The process of going through these states of an Android application and the possibility/impossibility of shifting from one state to another is the application lifecycle. Unlike traditional desktop applications, where the operating system takes care of all app states and changes among these states, the Android operating system cannot do the same.

The scarcity of mobile device resources makes it infeasible for the Android OS to store all states and state changes of all applications and whenever the state changes for these apps. This puts the burden of handling and managing the application data and state when the application is paused, swapped by another application on the phone or even shut down by the operating system. This means that ensuring optimal resource consumption, preventing data loss and reaction to application lifecycle state changes is the complete responsibility of the Android application itself [10].

An Android activity is a single action the user is able to perform. Android activity class creates a window to hold the UI of the app screen, so the user can interact with the activity. When the user interacts with the activity, the activity instance undergoes state changes. The changes of activity instance state represent the activity lifecycle. In order to be aware of and react to state changes of the activity, the activity class has callback methods, also known as activity lifecycle methods. These lifecycle methods indicate the Android operating system's process regarding the activity window, which is the current application screen. These lifecycle methods show whether

the Android system is creating, resuming or stopping an activity, or even killing the process of the activity.

Developers can specify exactly what should happen whenever the user enters or leaves the activity inside the any activity lifecycle callback method. For example, if the user is streaming a video within a video streaming application, and the user decided to switch to another application, it is the video streaming app responsibility to pause/kill the streaming process and cut off the network connection. Hence, a callback lifecycle method allows the Android app developer to state what should happen upon app state changes. It is very important to perform the right tasks and reaction to an application state changes. Using the lifecycle methods right can make the difference between a highly performant app and a buggy app that keeps crashing.

Android operating system offers six activity lifecycle methods. These methods are onCreate(), onStart(), onStop(), onResume(), onPause(), and finally onDestroy(). The first method *onCreate* is where the main logic of the application startup is coded. This callback method is triggered upon getting into the *Created* state of the activity. This logic only runs once per the activity lifecycle. The second method is the *onStart,* which shows the activity to the user and prepares it for interaction, and it is where the developer places the code to maintain the app's GUI. This method is triggered upon getting into the *Started* state.

The *onResume* method is triggered once the app gets into the *Resumed* state. This method is where the user can start interacting with the application. Once the app's activity is no longer in the foreground, and whether it's being destroyed or paused, the application enters the *Paused* state. This triggers the *onPause* lifecycle method. In this method, the developer codes what's expected to happen while the application is being shotly interrupted by a phone call for example, a multi-app

window is active and the focus is no longer on the current app, or even a dialog has opened on top of the current activity which makes our app's activity not fully visible.

When the current app's activity is no longer visible and completely covered by something else, or the app itself in the background and another application is now active, the activity enters the *Stopped* state. This state triggers the *onStop* lifecycle method. Inside this method, the developer can code logic involving operations of saving information to the database for example. The last lifecycle method is *onDestroy*. This lifecycle method is triggered right before destroying the activity. This method can be used either because the activity is finished, or because the Android system is killing the activity for some reason. In this method, the developer needs to clean up after the app components and activities and release all system resources [11]. The following diagram depicts the activity lifecycle methods and transitions between them.

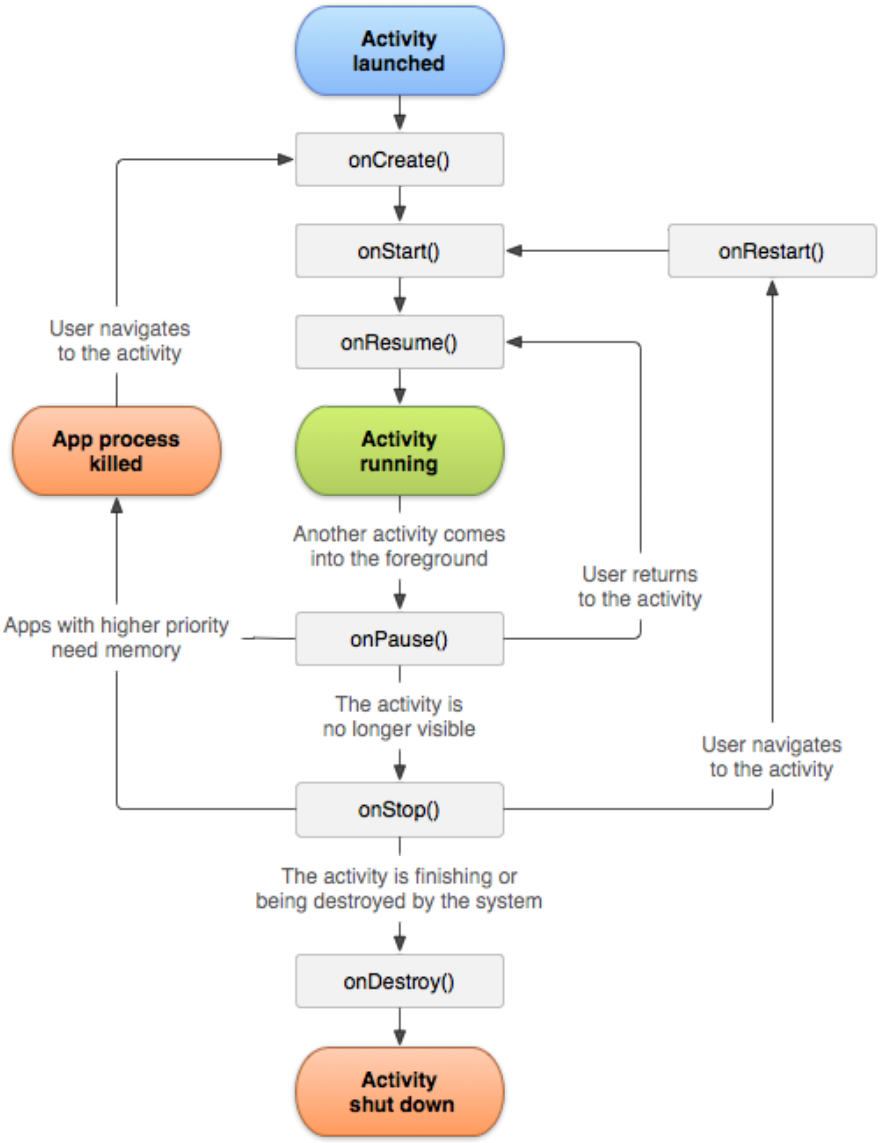


Figure 1: Android Activity Lifecycle Methods

Source: [11]

Chapter 3

Literature Review

3.1 Introduction

In our search for automated model-based test case generation for mobile applications, we came across three main categories of papers. First we start off by looking at the current state-of-the-art of mobile application testing and the available automated testing tools.

Secondly we look at the generic model-based testing category which discusses papers that look into generating test cases using predefined generic models. In the same area, we then discuss the advantages of using model based testing approach in mobile applications, or comparing model-based testing approach with other automatic test case generation approaches.

The third category of papers delves deeply into the model-based testing approach for mobile applications and conducts custom tuning of the steps it takes to produce a model for automatically generating test cases, the actual generation process of the test cases, the execution of the generated test cases and finally the evaluation of these executed test cases.

In this chapter, we go over these papers, discuss and compare the problems each paper discusses, the methodology they follow and finally the results these papers come up with.

3.2 Literature Review Methodology

GoogleScholar.com is the engine used to search for papers. We filtered through multiple academic journals and articles. These include Springer, IEEEXplore, ACM, ResearchGate, AST, ICITCS, ICSRS, and ICT4M. We also used Google's documentation for Android developer guide and some other online resources.

Main keywords and search strings used to elicit the papers from these resources include mobile testing, Android testing, lifecycle testing, automation tool, application lifecycle, Android

activity, automated testing, automation tool, automatic test case generation, model-based testing, Android GUI model, GUI testing, generation of test cases from model and automatic lifecycle test generation.

In order to eliminate researcher bias, inclusion criteria for selecting and presenting papers related to the research topic are as follows. For a paper to be included in the literature review, it has to be:

1. Has been published within the past five years
2. The paper is at least five-page long
3. Preference for papers that are empirical studies

3.3 Mobile Application Testing

Zein et al. [4] conducted an exploratory multiple case-study to try and understand the testing methods developers use in real world mobile applications and the obstacles those developers encounter while testing. This empirical study involved four mobile app development companies.

The authors concluded that in all studied cases, both developers and testing engineers do not have the expertise or full knowledge in testing methods or tools to create or test mobile apps which comply with Android lifecycle properties or models. Besides, the study found that testing engineers do not possess the knowledge to perform cross-application communication testing, which is called integration testing. Moreover, the study concludes that no official and systematic testing approaches exist to help test critical apps. In fact, most industry testing of mobile apps relies on the manual blackbox testing of GUI. Also, automated testing tools are rarely used to test

application lifecycle conformance. In short, mobile app developers main focus is to quickly produce responsive apps with fancy GUIs.

At the same time, Muccini et al. [1] present a generic overview of mobile application testing state-of-the-art, and its future research directions. The authors try to understand how mobile applications are different from traditional ones, and how this affects the types of needed testing methods. Also, they delve into the challenges and the directions research is heading towards for mobile app testing. Finally, the authors talk about the role of automation in the mobile app testing process.

The authors describe that automating mobile app testing is crucial for two main reasons. First, automation reduces the cost of testing and at the same time guarantees the quality of the apps under test. Second, ensuring Layer testing. This means testing the interoperability of applications through the operating system, among apps, and against device hardware too, such as sensors. Reported bugs indicate that problems arise due to issues between the applications and operating systems, and not only within the apps.

The same paper explains that achieving cost effective testing for the mobile app can be achieved by means of cloud-based testing and outsourcing. Besides, research indicates that the industry may be heading towards the services of testing as a service, which would yield affordable testing solutions for mobile apps.

3.3.1 Automated Mobile App Test Input Generation

Linares-Vasquez et al. [13] highlight in a survey the state-of-the-art of automation testing for mobile applications. The Authors cover the services, tools and frameworks that mobile app developers can use in app testing as well as presenting some of the drawbacks of these available testing methods.

The paper introduces GUI automation APIs and frameworks as one of the existing methods for testing mobile apps. These frameworks are often used to get an overview of the GUI components structure. Such tools are usually used by developers and testers to help write automation scripts that run by record and replay tools. These record and replay tools are used to record testing behavior, then generate test scripts that can be run later and even modified to cover different test cases. However, record and replay tools suffer from a compromise of either accuracy or timing of recorded and replayed events.

Another type of testing methods is automated test input generation. This is one of the most active research areas of automated mobile app testing. Research in this area presents several approaches for automated test input generation. These approaches include random testing, systematic testing, search-based testing, combinatorial testing, and finally model-based testing which is our focus in this research.

Error and bug reporting tools are another type of the commonly used testing methods nowadays. This type of tools usually comes in two types, the first is the regular issue tracker for bugs and the second is the crash and resource consumption monitor tools. This type of tools helps provide a textual description of the bugs as well as adding snapshots of the errors and resource consumption.

Mobile testing services are another type of mobile testing approaches. This approach relied on outsourcing the testing part of a mobile app to a group of testing experts and/or non-experts outside of the development company. This allows for less testing costs.

Tools for device streaming are also used to facilitate the testing process of mobile apps. This means that developers can mirror the mobile device to a personal computer screen or even remotely connect to that mobile device when needed.

Despite the fact that all of these automated testing methods and techniques exist, manual testing is still used more than automated testing for mobile apps. The authors explain that manual testing is normally preferred due to several factors including lack of the needed functionality in many of these tools, personal testing preferences by the testers, or organizational limitations in some of the cases.

Delving deeper into the available automatic test input generation tools, Choudhary et al. conduct an empirical study [15], by performing a comparison of tools of the existing automatic test input generation for Android devices. The authors evaluate how effective these tools are in terms of code coverage, fault detection capability, compatibility with Android platforms and ease of use.

When it comes to code coverage, the authors state that several methods are used which are systematic, random and model-based test input generation. The research states that it's not clear that any of those strategies is better than the others in practice. However, the authors explain that the most important factor when it comes to automatically generating test inputs is time. This means that all tools should focus on how much code coverage they can achieve within a limited amount of time to help measure the effectiveness of these tools. Monkey [16]; which is a random-based test input generation tool presents the best results in terms of code coverage.

The second metric the paper compared tools against is ease of use. This means that the tool needs to be effective in terms of working right away and out of the box, with not much configuration involved. The authors found that Dynodroid [17] and Monkey to be the easiest to use. While ACTEve [18], A^3E [19] and GUIRipper [20] need a lot of configuration to use.

Compatibility with Android platforms is another metric used to compare automation tools. This means that the test input generation tool needs to run on different Android devices with variable

hardware characteristics and Android versions. Monkey, GUIRipper, and ACTEve are compatible with all Android versions and devices.

The last metric measured in this research is fault detection capability. The authors measured this metric by counting how many bugs a tool can detect within one hour of running per app. Monkey tool was able to find the highest number of bugs within the set time limit in this case. Consequently, the research shows that Monkey tool depicts the best performance when according to the four benchmarks specified in this research.

3.4 Generic Model Based Testing

In the empirical study of de Cleva Farto and Endo [21], they conduct an experiment to measure the effectiveness of using model-based testing in generating test cases automatically for Android applications. They aim at checking if current model based testing can be used to test functional requirements for mobile applications. Besides, they try to identify the results and issues of using model based testing in mobile applications. Finally, they try to measure the effectiveness of the generated test cases on Android applications.

The authors use an event sequence graph (ESG) as the modelling method. They develop test cases using Robotium framework [22]. The research finds that model based testing is a valid and recommended approach for automatically generating test cases for Android applications. The authors found that model based testing is efficient in terms of generating test cases automatically, detecting faults in the system under test, good test case quality, reduced cost and time of testing and finally the maturity and evolution of the testing model.

Challenges of the model based testing are also presented. The study shows that modelling itself proves a difficult task, making the testing concrete for mobile applications in general and finally requiring experience in certain tools to perform the tests. However, the study concludes that

model based testing using an event sequence graph is an effective and systematic method for testing Android applications.

Saad and Abu Bakar [23] discuss selecting the proper testing tool for the mobile platform of choice and depending on the research requirements. They introduce a variety of automated testing tools for the mobile platforms of Android, iOS, Blackberry, Symbian, Windows Phone and Windows Mobile. The authors focus on the verification of methods that are needed to ensure the mobile app works as expected, so in other words they assert the functionality of the mobile app with generic blackbox tools.

Their criteria for choosing the right testing tool include how well the tool handles different web browsers, emulators, support for different operating systems, types of GUI testing they offer, and interruption testing abilities, test reporting capabilities, test workflows, and pricing. The chosen tool of the research is Micro Focus Silk Mobile, a one time payment tool. The authors mainly chose this tool for the support it offers on all platforms and for providing high quality test flows.

On the other hand, Frajtak et al. [24] introduce the challenge of using machine-aided exploratory testing rather than manual exploratory testing to generate test cases. The proposed approach is suggested where the system under test model is not available. This approach uses a reverse engineering method to recreate the model of the system under test.

This research conducts a case study with two groups of testers. Where one uses manual exploratory testing and the other uses machine-aided exploratory testing. The authors propose a testing framework to help with the exploration of the system under test.

The research finds that the exploratory testing aided by the proposed framework achieves better results in terms of documenting the testing process. The documentation mostly covers the

steps followed to perform a test case as well as generic documentation of the explored areas of the system under test. The authors measure the efficiency of this approach by comparing it with manual exploratory testing. The results of this comparison show that the machine-aided exploratory testing saves 23.54% more time than the manual exploratory testing.

Another commonly studied approach for test case generation that relies on equivalence classes, where Subramanian et al. [25] discuss partitioning of equivalence classes as an approach to generate test cases for Android application GUI. This is a manual approach that is based on specifying the functionality and the GUI specifications.

The proposed approach use class depends on the equivalence class coverage method. This method produces test cases for the GUI immediately and it fits early stages of the app development lifecycle. This approach adapts well to changes in the application, since it performs systematic exploration of the test cases. Besides, the proposed testing approach can help with app maintenance since unnecessary testing errors can be filtered out.

3.5 Specialized Model Based Testing

Amalfitano et al. [26] introduce the problem of automating the generation of GUI tests. They present MobiGUITAR as a tool for automated GUI testing of Android apps. MobiGUITAR is a run-time tool for observing, extracting and abstracting the GUI state of the Android app. This tool is based on an abstraction Model that has criteria for test coverage to generate unit test cases automatically. The tool relies on a reverse-engineered model for mobile apps. The authors of this tool apply it on 4 open-source Android projects in an empirical study to generate and run over seven thousand test cases and find 10 new bugs.

MobiGUITAR consists of three steps. First, it traverses the mobile app GUI in order to create a state-machine model (graph) for the GUI to be used for test case generation (also known

as GUI ripping). Second, MobiGUITAR generates test cases for the resulting GUI sequences of events. These test cases are based on the rule that pairs of adjacent events (edges on the graph) are grouped together to merge the humongous number of sequences of events for the app. Finally, the execution step outputs the generated test cases as JUnit-formatted outputs. This helps detect app crash bugs during run-time, which covers the IllegalArgumentException.

The authors conclude that the tool they created helps generate test cases that in turn help find severe bugs in the applications under test. Moreover, the authors depict that using model-based testing along with model learning offers improved fault detection in the realm of testing Android applications.

In another study, Espada et al. [27] executed a model-based testing approach to explain the different possible user interaction flows in mobile applications. The study is conducted by using a tool to explore the model generated by a custom finite state machine aimed at detecting all potential user interactions. The proposed approach uses model-based testing. The authors built a tool that uses a model to generate test cases. The generated test cases consider both user interactions with the applications and the applications interactions with each other. Then to analyze the expected behavior, the authors use a tool called SPIN to analyze a specially designed state machine to get all available user interactions corresponding to the generated test cases. Finally, the model generated test cases are run on an Android device to mimic user behavior.

Compared to the approach the authors followed in MOBIGUITAR [26], this research did not need to clean up the generated test cases and remove the infeasible ones. This is because this study separated the test case generation from the testing process. Also, the states resulting from the custom state machine are designed to be limited and compact. Besides, MOBIGUITAR is

applicable on one application at a time, while this research allows for testing multiple applications interacting with Android intents.

This research concludes by describing how the results generated in this case study make realistic test cases and not randomly generated test cases that require data cleaning. The authors are yet to verify the efficiency of this tool with a runtime verification mechanism to test the effectiveness and performance of this proposed approach.

At the same time, Salihu et al. [28] presents a tool called AMOGA. This approach is proposed to overcome the issue of generating a model that automatically produces dynamic and comprehensive GUIs for mobile applications. This approach is a combination between static and dynamic methods of model generation.

AMOGA has a static analyzer and a dynamic crawler. The static analyzer extracts the event sequences that the mobile app supports. The dynamic crawler then crawls through these generated events and builds up the model of the mobile applications. AMOGA is applied to 15 Android applications in an experiment. The results show that AMOGA efficiently generates a thorough model which presents high code coverage of the system under test. Besides, mutation testing is applied to measure the ability of AMOGA to detect faults in the mobile app. The proposed tool achieved a good mutation score which proves that AMOGA can reveal several bugs in the mobile app under test.

In another study, Liu et al. [29] introduced the challenge of automatically generating relevant input text for mobile applications. They use a deep learning mechanism to automatically generate text input for 50 iOS mobile apps. The deep learning techniques are tuned to work on trained and untrained mobile apps for the deep learning model.

The authors claim to the best of their knowledge and at the time of this research, that they're the first to use deep learning to generate text inputs automatically for mobile apps. In short, their deep learning approach consists of two phases. The first one is the training phase. In this phase, the automated tool is trained by learning the manual inputs for testing and by associating these inputs with the relevant context of testing. The second phase is prediction. The automated tool predicts the input text, depending on the context.

After that, the research delves into evaluating the proposed approach in terms of effectiveness against other automatic generation of input for mobile. This evaluation involves comparing a Recurrent Neural Network (RNN) model they build with a random input generation approach in terms of performance and effectiveness in the scope of multiple cases studies. These case studies involve 50 iOS devices where the authors test the FireFox and Github iOS apps.

Finally the research compares the deep learning's RNN model with the Word2Vec algorithm in terms of performance and effectiveness. Eventually, the authors find that the deep learning approach for automatic input generation produces relevant text input in terms of the program context. The evaluation of the 50 iOS apps using the RNN model shows that the proposed RNN model provides efficient and effective results for generating inputs.

Gudmundsson et al. [30] test the effectiveness of model-based testing to QuizUp Android application. This application represents the largest quizzing app on the market with millions of users around the globe. The study depicts that the model-based method which relies on a simple finite state machine can effectively and efficiently be used to test such huge Android apps. After applying model-based testing techniques on this app, the authors were able to detect major defects in the mobile app under test. These defects were then fixed and deployed to the QuizUp Android application.

In another study, Panizo et al. [31] introduced a model-based testing framework to automatically user interactions with the mobile app. The authors extend the TRIANGLE[6][1] tool for automatic model-based testing which relies on model checking mechanism.

This study uses the extended tool for testing the ExoPlayer Android application in several various network scenarios. This mobile application is a video streaming app that uses several streaming protocols. The major feature in the proposed testbed extension is that it emulates realistic networking scenarios that cover several configurations for network and radio. Integrating this feature into model-based testing resulted in better test coverage for user flows, the ability to further extend the model with additional user flow conditions without changing the model but rather defining new rules for it and the simplicity of defining testing criteria within the tool with plain language for average developers.

At the same time, Frajták, Bures and Jelinek [32] present a hybrid testing methodology that combines the model-based testing approach along with the manual exploratory testing. The authors present this combination to try and eliminate the issue of verifying and documenting the resulting test cases when the model for test case generation is incomplete or inconsistent and we need to re-evaluate its results or measure its testing effectiveness. The model of test case generation is dynamically created and updated in the exploratory stage. Each step for the test case generation in the exploratory model is marked via a JavaScript tracking code. This tracking code is injected in web pages that represent the channels of communication for the test case input data.

The study conducted an experiment where two groups, one used the manual exploratory testing and the other used model based testing. The research found that manual exploratory testing had the advantage in some of the sub-tasks of providing documentation of the testing flow, re-evaluating the testing scenarios, and documenting the explored and non explored areas of the

[1] https://www.triangle-project.eu

system under test. This is an advantage of manual exploratory testing that model-based testing does not have that this research found.

In the study conducted by Zhang, Wu, and Rountev [33], the authors look into testing Android applications to explore and find the leaks and defects in resource usage. They used a static analysis model to define the regular GUI flows that have a normal effect on Android app resources. After that they introduced a test input generation algorithm to generate these normal flows and then categorize these flows into two categories.

The categorization of the generated outputs of these algorithms followed the patterns of resource leak of the Android applications under test. After that, the authors compared these algorithms of automatic test input generation with non automated algorithms that the authors have presented in previous work.

The result of this study indicates that it is actually possible to automatically generate effective and generic test cases for detecting leaks in Android app resources using the methodology they propose.

3.5.1 Custom-tuned GUI Test Generation

Baek and Bae [34] introduce automated GUI testing using a model for test case generation as well as debugging. The authors follow a systematic approach in an empirical study to understand the effect of multilevel criteria of GUI comparisons on the effectiveness of testing.

The authors introduced the multilevel GUI comparison criteria (GUICC) framework as a GUI model generation methodology which focused on the way GUI model is generated for Android applications. They conducted an empirical study to test the effectiveness of GUICC for testing the effectiveness of Android GUI testing models. They tested the framework in terms of

research questions about the effectiveness of the generation GUI graph, the code coverage that graph offered and finally the error detection capability of the GUICC framework.

As a result, the authors found that multilevel GUI testing achieved more effective activity based (single level) GUI modelling and testing. Besides, they found that state explosion issues can be significantly minimized with multilevel GUI comparison criteria, which is an issue in single level GUI testing.

The authors discuss the issue of implementing automated GUI testing approaches and comparing the outcome of each of these approaches. The authors present a generic testing algorithm in the context of a conceptual framework. The aim of their framework is to use it for building up methods to automatically test GUIs and compare the outcomes. The framework itself is based on the generic and configurable algorithm that can be tuned to produce several various Random testing and Active Learning testing methods via modifying the six input parameters of the algorithm.

In a different study, Amalfitano et al. [35] applied a specially designed conceptual framework on an Android application. They defined and applied nine testing approaches by configuring the AndroidRipper [3] in nine different ways, which resulted in nine different testing approaches. After that, they compared between the results for each of these approaches. The goal of this comparison was to try and understand how the change of the algorithm's six inputs affected the adequacy of testing, the cost and the resulting GUI tree complexity for each of these approaches. Six of these nine testing approaches use Active Learning strategies, where model learning and GUI testing are used together to learn the model of the GUI to generate events/test cases based on the inferred model.

The research concluded that descriptive testing strategy, where the analyzing the GUI is described in relevant component subsets, deeply affects the performance of the testing approach as well as the testing model complexity. Whereas the scheduling strategy, where the schedule for firing next events is provided, didn't. In fact, the models built using both the active learning and non active learning methods are biased when built using the scheduling testing strategy.

3.6 Android Activity Lifecycle Conformance Testing

Zein et al. [39] introduce an automated testing approach using static analysis to help junior developers keep track of used system resources during the Android applications' lifecycle. The difference between my research and Dr. Samer's is the following:

- Dr. Samer's work is merely based on static code analysis, while mine is model based. I build a model and the output of every phase is previously clarified in the methodology chapter.
- My model-based tool checks activity files, fragments and AppCompatActivity files, while Dr. Samer's only checks activity files.
- My model-based solution approach (tool) generates text files for each activity, fragment and AppCompatActivity file, and these text files represent graphs of the lifecycle methods mode for that activity, fragment or AppCompatActivity. These graph representations contain the lifecycle method transitions from one lifecycle method to the next and the resources acquired/released inside each lifecycle method. Dr. Samer's static code analysis doesn't have such phases or lifecycle method representations for checking resource acquisition/failure in the next phases.
- I conduct a user acceptance experiment to verify the benefits and usability of my model-based solution approach, while Dr. Samer doesn't

- My model-based solution approach injects the *TODO* comment recommendations inside the Android apps' source code, to help the developer locate the resources that fail and where to release these resources, Dr. Samer's tool doesn't offer such functionality.

3.6 Summary

In conclusion, we discussed work related to automatic testing of Android applications. As shown in the previous sections, research in this field usually focuses on depicting state-of-the-art mobile application testing and the available automated testing tools in the market.

In other studies, research focuses on generic model-based testing which discusses papers that look into generating test cases using predefined generic models. In the same way, some papers tend to discuss the advantages of using model based testing approach in mobile app testing, or comparing model-based testing approach with other automatic test case generation approaches.

Another category of papers delves deeply into the model-based testing approach for mobile applications and conducts custom tuning of the steps it takes to produce a model for automatically generating test cases, the actual generation process of the test cases, the execution of the generated test cases and finally the evaluation of these executed test cases.

We elaborated on these approaches and fields in the above sections in terms of the problems they tackle, the methodologies used to tackle these problems and finally the results each paper discovered. However, no research focuses on generating lifecycle tests automatically for Android activity lifecycle callback methods. This is where we shed the light and conduct our research to fill this research gap.

Chapter 4

Methodology

The methodology we propose is building a model of the Android lifecycle callback methods, then use this model to generate test cases and finally execute these test cases and analyze the results. The lifecycle model will be the base of our test case generation process. The following figure depicts our model and its four phases.

4.1 The Model

The following diagram summarizes the main phases of the proposed model.

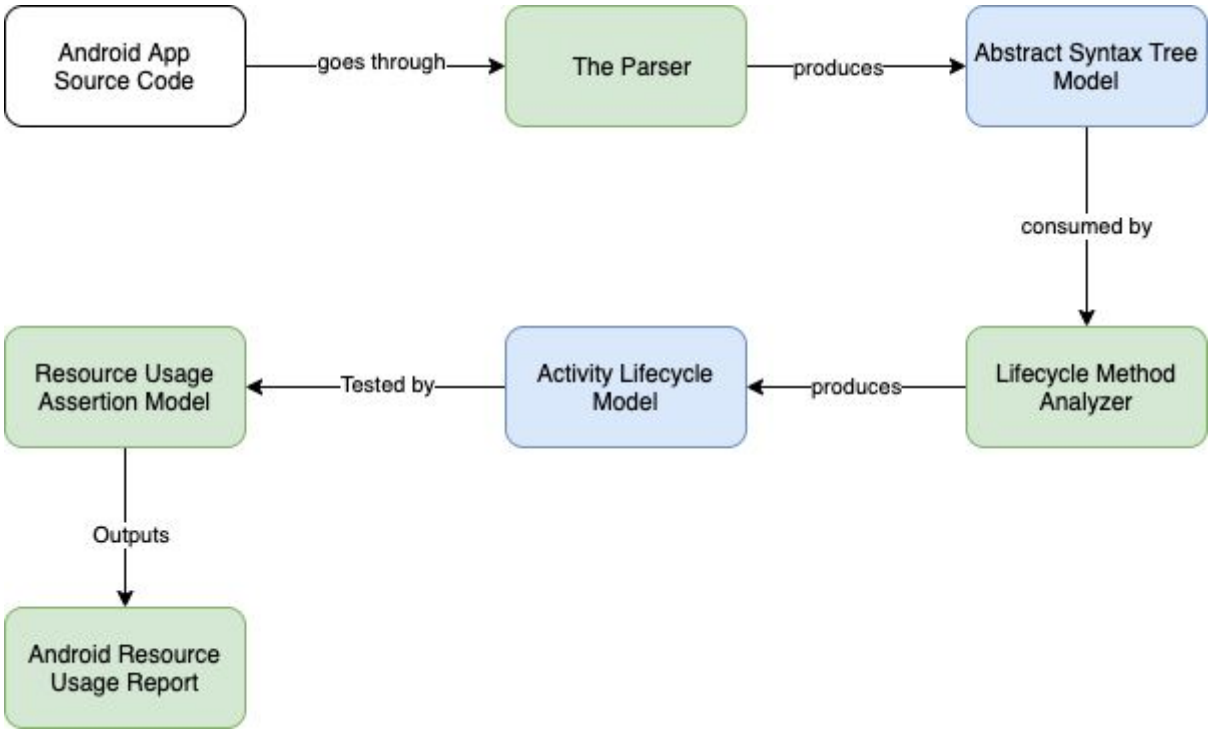


Figure 2: Model Phases

This model-based solution approach (tool) is implemented using the language Java, version 8. The implemented tool which is based on this model follows the phases and input and output

flow described in the next sections for each phase of the model. The model-based proof-of-concept tool we present checks Android app activity files, fragments and AppCompatActivity which is used for backward compatibility features.

4.1.1 The Parser

This is where the Android source code is first consumed. This phase takes the Android app source code and parses each of the activity files in this app using static code analysis. The parser parses one activity at a time, until all activities are parsed. The parser will only parse the Android lifecycle Lifecycle methods within each activity. The output of this phase is an Abstract Syntax Tree (AST) model.

The parser used in this phase is the Java Parser [37]. This publicly available library enables us to interact with the source code to be parsed in Java object representation format. This object representation format is called Abstract Syntax Tree (AST). This Abstract Syntax Tree data structure helps navigate and traverse the parsed code conveniently.

The example below demonstrates what the Abstract Syntax Tree representation is for Java code that prints the time [38].

```
package com.github.javaparser;

package java.time.LocalDateTime;

public classTimePrinter {

public static void main(String args[]) {

    System.out.print(LocalDateTime.now());
```

```
    }
}
```

The time printer code above is parsed and represented in the high level Abstract Syntax Tree shown in the following figure

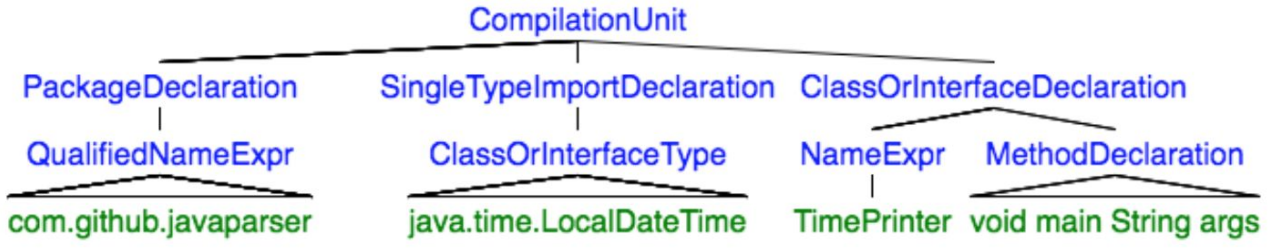


Figure 3: Abstract Syntax Tree [38]

The Abstract Syntax Tree Model in this phase resulting from parsing the Android source code is then consumed by the Lifecycle Method Analyzer in the next phase.

4.1.2 Lifecycle Method Analyzer

This phase of the model consumes the Abstract Syntax Tree resulting from the previous phase and uses the AST to build a graph for each activity of the Android app. Each graph represents the activity lifecycle methods and the transitions between them. Each node in the graph contains a lifecycle method in that activity and the resources it uses (catches/releases). Each edge

in the graph represents transitions between the lifecycle methods. For example, Camera resource: where it was opened (onResume), and where it was let go (onPause).

The figure below shows a sample graph with two lifecycle methods and the resources acquired and released in each of them.

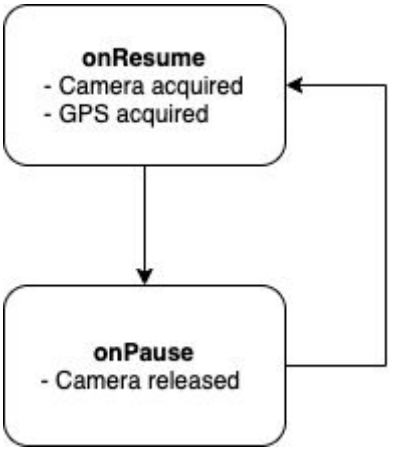


Figure 4: Activity lifecycle methods graph

This phase of the model produces a directed graph for each activity file in the Android app and the lifecycle methods inside that activity file, similar to the graph in figure 4. In order to help visualize and see the results of this phase, the graph for each activity is printed on a separate text file and inside a folder named ‘log’, and the output on that text file is printed in the following format:

onResume()	--->	onPause()	--->	onResume()
- Camera acquired		- Camera released		- Camera acquired
- Location acquired				- Location acquired

This activity lifecycle graph is then consumed by the assertion algorithm in the next phase of this model to determine the success or failure for each of the acquired resources.

4.1.3 Resource Usage Assertion Model

In this phase, we build lifecycle test assertions to make sure each resource, e.g. the camera was acquired and released in the relevant lifecycle methods. These assertions tests take into consideration three resources and a set of predefined model rules for each resource to use for assertion. In the model-based tool we implement assertions for three system resources; the camera, the location and finally the external drive. We chose these 3 resources because they're the most commonly used system resources by Android apps.

The first resource is the camera. This resource is checked by searching all lifecycle methods for the camera object instance, which acquires the camera. This object is usually initialized inside the onResume lifecycle method. However, the model checks for instances of the camera object in all lifecycle methods in case the camera was acquired in any of them. The camera in Android (version 11) can be acquired in one of two ways:

1. Directly using a camera object instance. This camera object instance can be directly initialized in the lifecycle method itself, or inside another regular method which then is invoked inside the lifecycle method, called *cameraInstance* for example. Afterwards, this camera object instance is used to acquire the camera resource using *cameraInstance.open()*. Normally the camera is released inside the onResume method, but we check the other lifecycle methods as well. We check if the camera resource was released by looking for the *cameraInstance.release()*.

2. Using the Android camera manager which uses the camera via system services. We check for an instance of the camera manager object, called *cameraManager* for example. In order to check if the camera manager instance is released or not, we look for the *cameraManager.close(),* which should be in the onPause lifecycle method.

The second resource we check is the location. We check for location acquisition in lifecycle methods (usually in the onResume method) and we look for an instance of the *FusedLocationProviderClient* and if it invokes the *getFusedLocationProviderClient* method. In order to check if this resource is released, we look for the *removeLocationUpdates* inside the onPause lifecycle method.

The third resource we check in this model is the external storage (drive). We check for reading from or writing to external storage using either the Android *ParcelFileDescriptor* or *InputStream/OutputStream,* depending on whether the media content is best represented as a file descriptor or a file stream.

The model-based solution approach (tool) we build has a property file used to define resources and the rules for each one of these resources. The property file which is managed by the developer consists of the following attributes and methods as inputs to the tool:

- Name for the resource we're defining. For example, we alias the camera resource caught using the camera object instance 'Camera' and the camera resource caught using the Android camera manager (via system service) 'Camera2', in order to clearly show the resource that failed and how it was acquired.
- The way the resource is caught. For example, we define *cameraInstanceVariable.open()* as the way 'Camera' resource is caught using the camera object instance. While we defined

getSystemService(Context.CAMERA_SERVICE) as the way the camera is caught using the camera manager.

- Lifecycle methods where the resource can be acquired. For example, we define *onResume* as the method where the 'Camera' and 'Camera2' can be acquired.
- Lifecycle methods where the resource can be released. For example, we define *onPause* as the method for where the camera resource can be released, in both camera definition methods.

4.1.4 Resource Usage Report

The final phase of this model is to present the assertion results for the checked resources. The model checks for resource acquisition/release using the assertion rules and for each of the 3 resources described in the previous section. If a resource is caught, but not released, then the test fails. The model-based tool we present provides details about each resource and in what activity it was caught (acquired), and whether this resource was released (pass) or not (fail).

Figure 5 below shows a screenshot of the results window of the tool after analyzing a sample Android app and showing the parsing and checking results for 7 activities and their associated resources. We notice the results colored red, which are 1, 4 and 5 are failing resources because the related resources were acquired but not released. The tool shows the failing resources and the recommendation to release the failing and in the appropriate lifecycle methods.

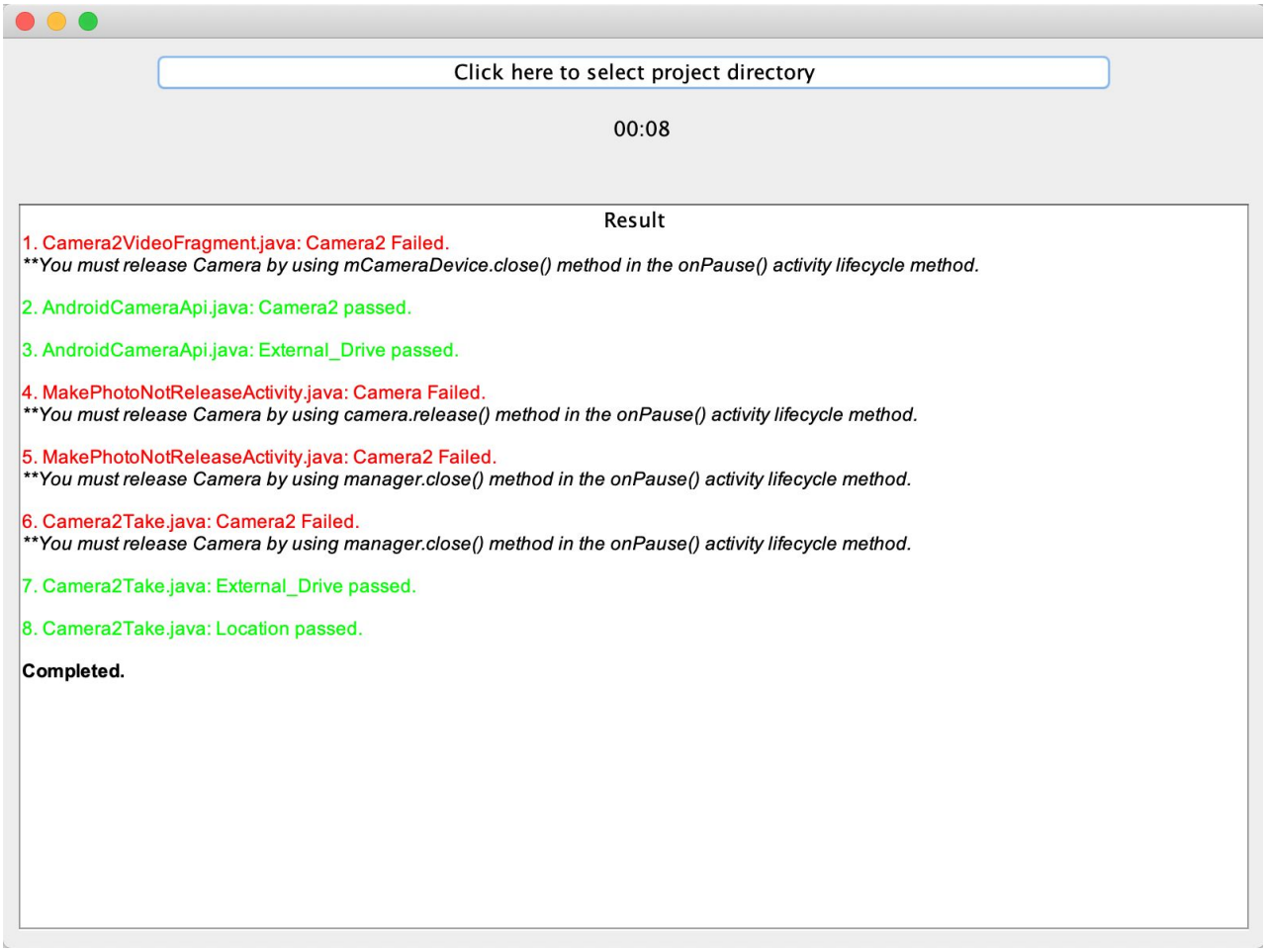


Figure 5: Sample tool Analysis Results

Besides, this model-based tool we present inserts a *TODO* comment in the Android app's source code and in the activity file to help the developer locate the resource that failed the test. The comments are inserted for resources that fail, and two *TODOs* are inserted for each failed resource:

1. One comment is inserted above the line of the instance of the resource that acquired the system resource. Screenshots below show the *TODOs* for the camera resource acquired once using camera object instance, and once acquired using camera manager, and neither were released.

```
// TODO: You must release Camera by using ###.close() method in the onPause() or in onDestroy() activity lifecycle method.
CameraManager manager = (CameraManager) activity.getSystemService(Context.CAMERA_SERVICE);
```

Figure 6: sample *TODO* comment injected above the code acquiring camera using camera manager

```
// TODO: You must release Camera by using ###.close() method in the onPause() or in onDestroy() activity lifecycle method.
CameraManager manager = (CameraManager) activity.getSystemService(Context.CAMERA_SERVICE);
  // TODO: You must release Camera by using ###.release() method in the onPause() or in onDestroy() activity lifecycle method.
  if (cameraId < 0) {
      Toast.makeText(this, "No front facing camera found.", Toast.LENGTH_LONG).show();
  } else {
      camera = Camera.open(cameraId);
  }
}
```

Figure 7: sample *TODO* comment injected above the code acquiring camera using camera object instance

2. A second comment is inserted on top of the activity lifecycle method where the resource was acquired. Figure below shows the *TODOs* injected above the lifecycle method where the camera resource was acquired in 2 different ways (hypothetical scenario) to show how acquiring multiple resources in the same lifecycle method and not releasing them would be treated by the tool. These *TODOs* would summarize the failures for the developer and save some time to go look inside each of the invoked resource acquisition methods inside the lifecycle method to look for which resources are failing separately as per previous step when there are multiple failures.

```
/*
 TODO: You must release Camera by using ###.release() method in the onPause() or in onDestroy() activity lifecycle method.
 TODO: You must release Camera by using ###.close() method in the onPause() or in onDestroy() activity lifecycle method.
*/
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    // do we have a camera?
    if (!getPackageManager().hasSystemFeature(PackageManager.FEATURE_CAMERA)) {
        Toast.makeText(this, "No camera on this device", Toast.LENGTH_LONG).show();
    } else {
        getCameraAccess();
        openCamera(mTextureView.getWidth(), mTextureView.getHeight());
    }
}
```

Figure 8: Multiple *TODOs* for multiple resources for the lifecycle method.

These *TODO* comments help the developer locate the resources that are acquired and not released, so the developer can release these resources and avoid bugs.

Chapter 5

Evaluation

We evaluate the results of our research using two approaches. We first conduct a user evaluation test, an approach used before by Barnett et al. [36]. Secondly, we seed bugs into 10 real open source Android applications, then run our tool on these applications. This method was applied by Zein et al. [39].

We first conduct the user evaluation test to help measure user acceptance of the developed tool and if it could actually help make the development process faster and less buggy. The GitHub repository[2] of the project contains the entire automated testing application as well as an executable JAR file for the tool.

5.1 User Evaluation Experiment

We asked 6 Android developers to fill out a demographic survey prior to starting the experiment. This helped identify the experience level of the developers undergoing the experiment. After that, the participants watched a short learning video on how to use the automated testing application (our model-based tool) on a provided sample application. The learning video is so that we reduce the bias and at the same time run the experiment for multiple users, instead of going after them one by one.

The experiment setup went as follows:

1. Fill out a pre-experiment demographic survey (can be found in the appendix)
2. Watch a learning video for how to use the automated testing application.
3. Complete the programming task described below.

[2] https://github.com/MalikMotan/AutomatedAndroidAppTester

4. Finally fill out this evaluation survey.

5.1.1 Programming Task

Now once you've filled out the initial survey and watched the video on how to use the automated testing application, your task is to run the automated testing application on the sample Android application provided in this link and using the following simple steps:

1. Download the automated Testing application (Jar file)[3].
2. Download (or fork) the sample Android application[4].
3. Run the automated testing application and select the sample Android application as the input.
4. Check the successful/failed resources on the testing application's GUI.
5. Go to the Android app's source code and release the non-released (failed) resources in each of the corresponding activity files mentioned in the applications GUI.
6. Run the automated testing application again on the Android app.
7. Verify all resources now pass the tests.

Once the developer has taken the development task, he/she is asked to fill out the post experiment survey to measure the user acceptance. In the next chapter, we introduce the results and discussion. The full experiment surveys, survey results, sample app, learning video and executable JAR file are all in one public GitHub repository[5]

[3] https://github.com/MalikMotan/ExperimentAndEvaluation/blob/master/Android-Parser.jar
[4] https://github.com/MalikMotan/ExperimentAndEvaluation/tree/master/SampleAndroidApp
[5] https://github.com/MalikMotan/ExperimentAndEvaluation

5.2 Open Source Apps

We evaluate our automated testing application against 10 open source real world Android applications. These 10 applications are from different domains and have different sizes and complexities. We seed bugs into these applications and check the ability of our tool to detect these bugs. Table 1 below shows a summary of these applications.

App Name	App Type	Description	Lines of code in main activity
FooCam (200ms)	Multimedia	Take several consecutive shots with different exposure settings	297
AntennaPod (1.5s)	Multimedia	Flexible and easy to use podcast manager	514
CoCoin (1.2s)	Financial	Multiview accounting and financial management	770
LeafPic (1.6)	Multimedia	Material-designed fluid image gallery	633
Keepass2Android (1.1s)	Utility	Password manager to store and retrieve passwords	287
Camera2Basic (0.4)	Multimedia	Camera tutorial that is used to learn how to use camera	1036
Location Samples (0.3)	Navigation	Location samples library for best practices of utilizing location	241
OpenCamera (1.5)	Multimedia	Multifunctional and rich camera app	3038
Telegram (15s)	Communication (15s)	Messaging app with	4193

		high speed and security for exchanged messages	
WordPress (14s)	Productivity	Content management system for blogs and personal websites	1546

Table 1: The open source apps used for evaluation

The evaluation process for the 10 applications aimed at evaluating our tool's (automated testing application) ability to detect the correct and incorrect release of the acquired Android system resources. The evaluation process consists of three phases.

5.2.1 Phase 1

we first manually check the source code of all applications under test (AUT). This is to make sure these applications don't have lifecycle method errors. We check the imports of utilization of system resources to check which resources are being used. After that, we check the acquisition and release of each system resource to check if that resource is invoked in the right lifecycle methods.

5.2.2 Phase 2

This phase aimed at evaluating the wrong releases of system resources. In this phase we modified the source code of each of the applications under test in order to inject bugs into the main activity of each of them. Then we checked if our automated testing application can detect these injected bugs and display related errors on the application GUI. Our automated testing application tests for three resources, which are the camera, the GPS (Location) and the external storage

(drive). Our testing process includes scenarios for testing, which test resources incrementally. These scenarios are:

a. First scenario: we inject the error of incorrect release of the camera resource, then run our tool to detect this error.

b. Second scenario: we inject a second error to the camera one, by having incorrect release of the GPS (location) resource, then run our tool to detect these two errors (camera and GPS resources).

c. Third scenario: we inject a third error for the external storage resource, then run our tool to try and detect all three resources together (camera, GPS and external drive).

In this phase, we rely on manually modifying the Android apps' source code in order to inject bugs into them. We make sure each of the resources is acquired but not released. The incorrect release of system resources is done in either one of two ways, which are the common mistakes of junior developers when building Android apps. These two ways are:

1. Deleting the *release* method for each one of the three Android resources. For the camera resource, we delete the *cameraInstance.release()* method invocation for the camera invoked using camera instance method, or delete the *cameraManager.close()* method invocation of the camera using camera manager method. For the location (GPS), we delete the *removeLocationUpdates* method. For the external drive resource, we delete the *ParcelFileDescriptor* or *InputStream/OutputStream,* depending on whether the media content is best represented as a file descriptor or a file stream.
2. Inserting the release method for each resource into the wrong lifecycle method. For example, releasing one or more of the resources inside the onStop lifecycle method.

This phase also includes inserting the *TODO* recommendation comments where each failed resource is acquired. This *TODO* intends to help identify in the source code of the Android application where each resource was captured but not released. Then insert a comment above the line of code that acquires any of the three resources (camera, GPS and external drive), to recommend releasing that resource using the appropriate released method, and in the appropriate lifecycle method as mentioned in figures 6, 7 and 8 in the previous chapter.

5.2.3 Phase 3

The purpose of this phase was to evaluate if the automated testing application can correctly identify the appropriate release of system resources and in the right lifecycle method. In this phase, we aimed to check if our automated testing application can detect for each of the three resources if the resource was acquired and released in the right lifecycle methods.

In order to evaluate the execution performance of our automated testing application [40], we measured the time (in milliseconds) it took to analyze, parse and check the resources for each of our 10 applications under test.

Chapter 6

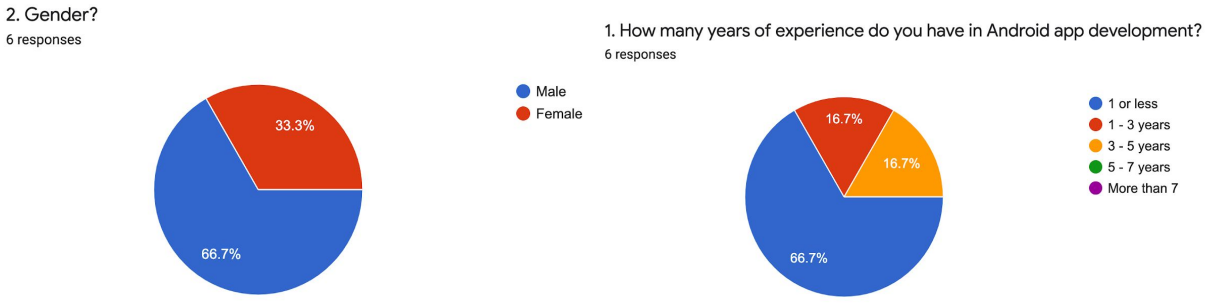
Results and Discussion

6.1 User Evaluation Experiment

6.1.1 User Evaluation Experiment Setup

There were 6 participants in the automated testing application's evaluation experiment who successfully completed the experiment. All of the participants are software developers with Android development experience levels ranging from one to five years. Two of the participants were female and four were male. The vast majority of those developers have built one to five Android applications in their lifetime. Only one of those developers has built 10 or more Android applications.

Five of those developers have bachelor's degrees, and only one of them has a master's degree. Almost none of those participants has ever used an automated testing application for their Android applications. It took the participants about 15 minutes each to finish testing the application and fill out the surveys. Figure below shows the responses to each of the demographic questions.



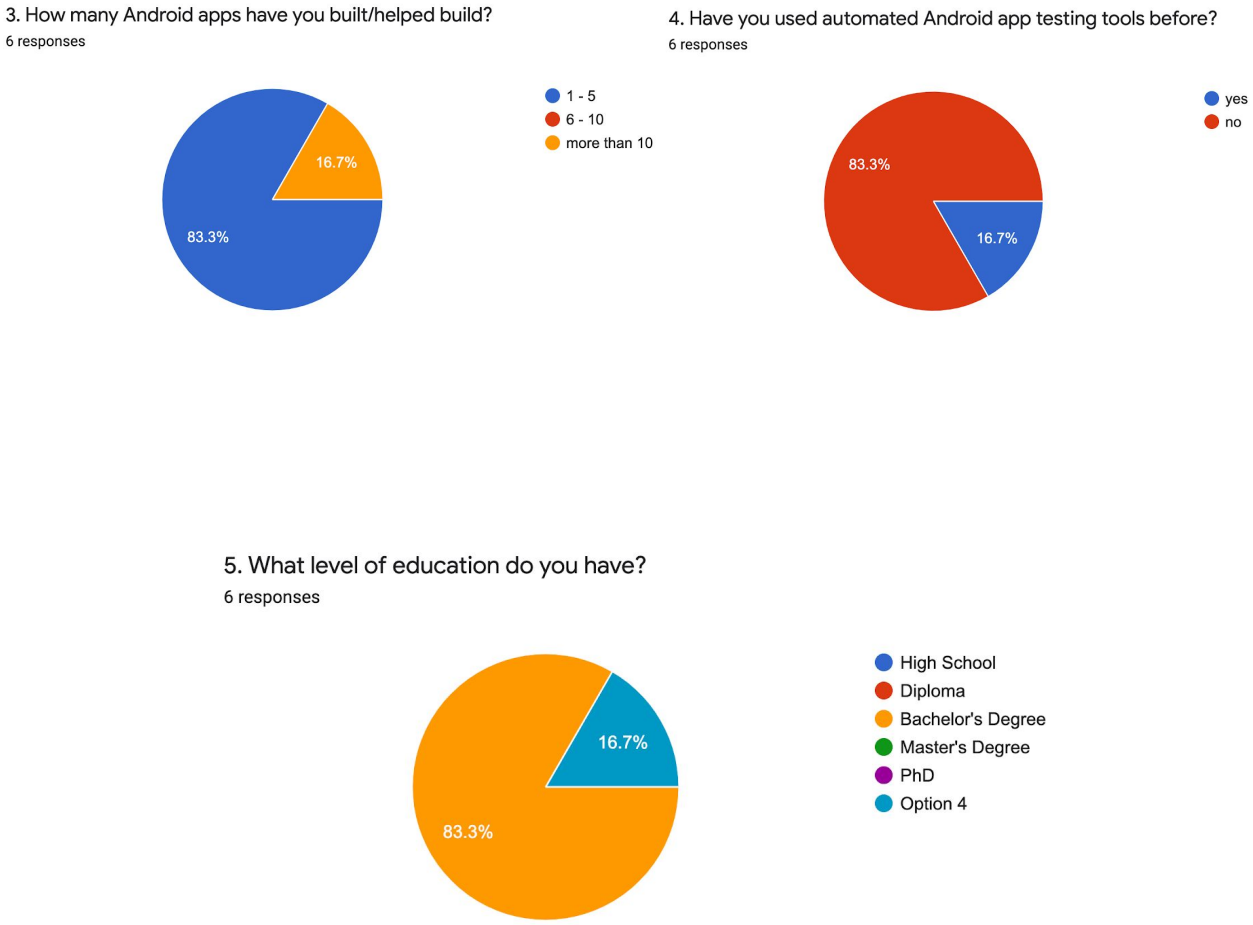


Figure 6: Demographic survey results

6.1.2 User Evaluation Experiment Results

When it comes to the post experiment survey, which intends to measure user acceptance, the vast majority of participant responses were positive. Around 90% of all participants either agree or strongly agree to 14 out of 15 to the likert scale type of questions. These strong results confirm that the proposed model and model-based tool is suitable for usage by professional Android application developers.

About 85% of participants agree or strongly agree that the automated testing application is fairly easy to use and easy to learn how to use it. Besides, that same percentage of participants strongly agree that it was easy to find the *TODO* comments inside the Android app's source code where the resources were acquired but not released. This in turn has helped those developers perform the programming task easily and fix the failing resources and run the automated testing application again to verify these resources were released properly and with no failures. Keeping in mind that those participants also found it easy to remember how to re-run the automated testing application when running it the second time and without making any errors.

Questions 3, 5, 6, 7, 11, 14 and 15 got 100% of participants to strongly agree on the following respectively:

- It was easy to find the log text files that represent the lifecycle methods model for each activity,
- It was easy to find the resources that were caught (acquired) but not released in any activity lifecycle method,
- The GUI of the automated testing application provided useful information about the resource the Android app was using,
- The GUI of the automated testing application helped identify the resources which were acquired but not released,
- The participants are satisfied in general using the automated testing application for testing their Android apps for resource failure and would recommend this automated testing application for fellow Android developers.

Furthermore, 85% of participants agree or strongly agree that the automated testing application helps make the development and testing process of Android apps faster and more productive. When the participants were asked in the open-ended question about what they liked the most about the automated testing application, two major points were raised:

- The success and failure of acquired resources.
- The *TODO* comments in the Android app's source code to help release the failing resources and in the relevant lifecycle method.

Finally, participants included some points regarding what features they would like to be included in this automated testing application for Android apps, these consist of:

- Checking all Android system resources in addition to the 3 resources the automated testing application currently checks.
- Releasing the failed resources automatically.

6.2 Open Source Applications Evaluation Results

The results of phase 2 evaluation for the 10 open source apps, where we check if our tool can detect incorrect release of system resources, were promising. The automated testing application successfully detected all incorrect releasings of the Android system resources and for all 3 scenarios. Figure 9 shows a screenshot of the execution results for the automated testing application for the AntennaPod Android app. The results of execution show that the external drive resource in one of the major activities failed the test, since it did not release the external drive in the right lifecycle method (onPause method).

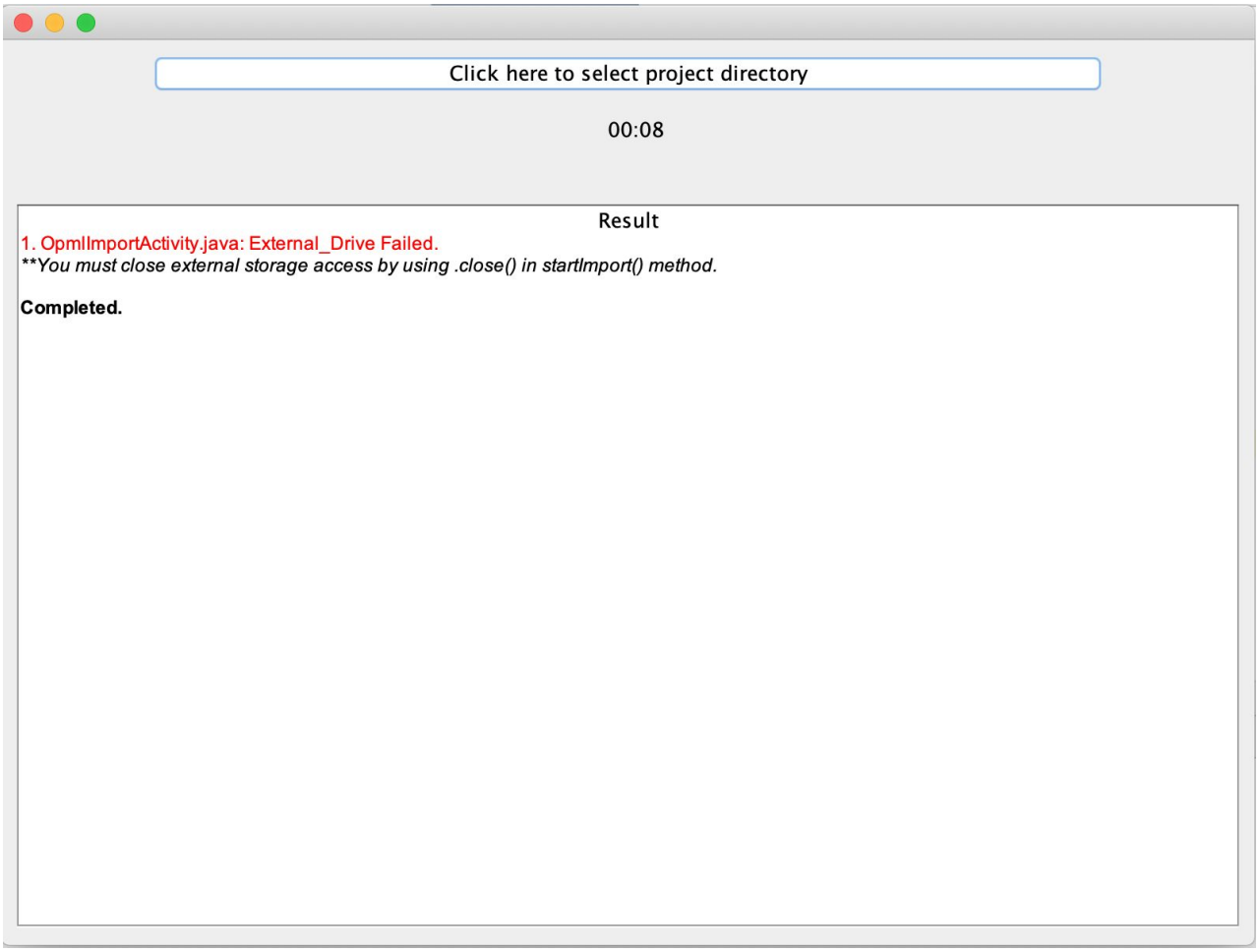


Figure 9: Execution Results of the Automated Testing Application for the AntennaPod App

When it comes to phase 3 testing, where we tested if our tool can identify correct release of resources, the results were promising too. The automated testing application successfully detected correct releasings of the Android system resources and for all 3 scenarios. Figure 10 shows a screenshot of the execution results for the automated testing application for the FooCam Android app. The results of execution show that the camera and external drive resources in the main activity have passed the test. This is because both the camera and the external drive were acquired in the onResume and released in the onPause lifecycle methods correctly.

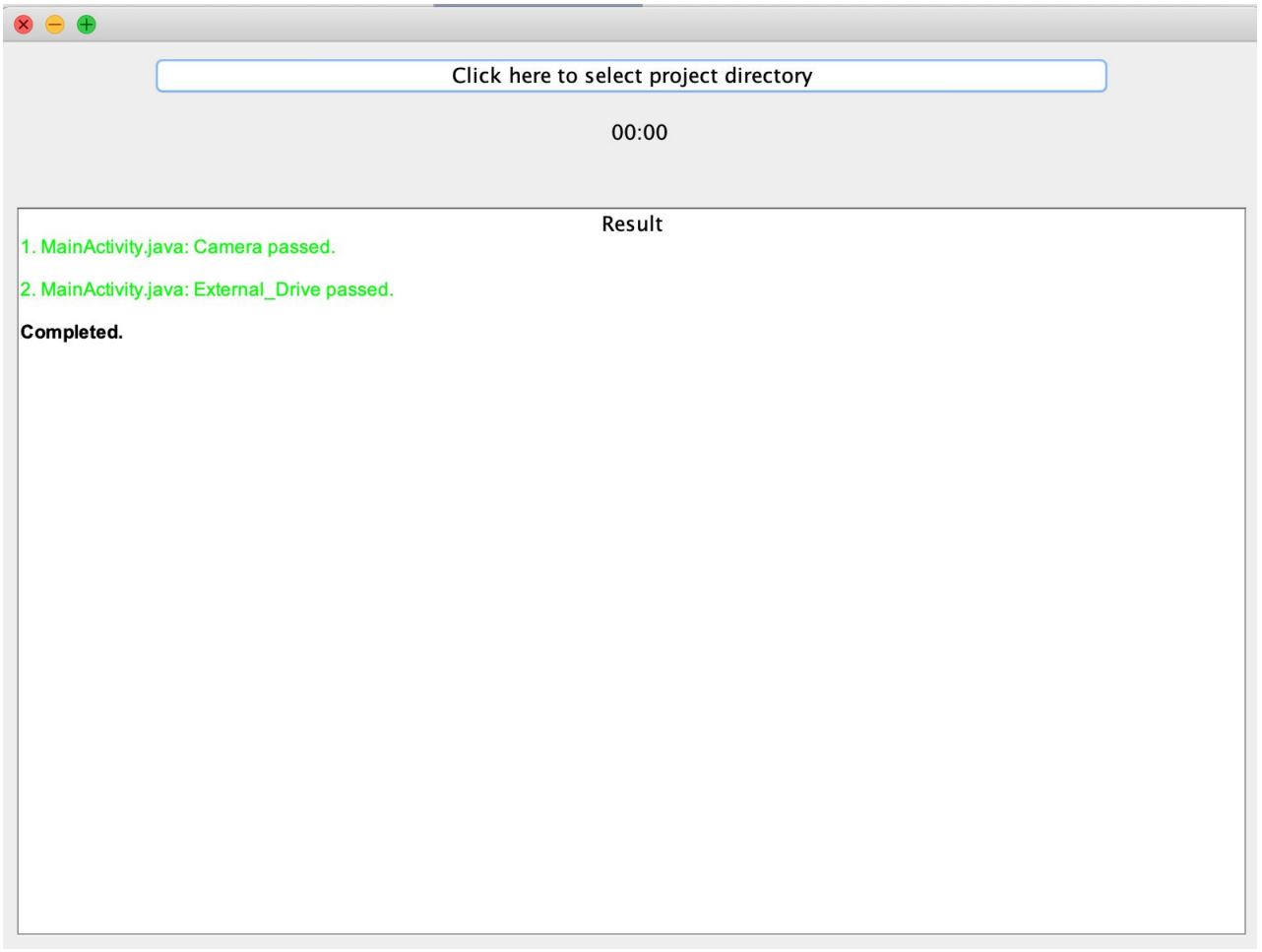


Figure 10: Execution Results of the Automated Testing Application for the FooCam App

As for performance evaluation of our automated testing application, we measured the execution time in milliseconds for each of the applications. Table 2 below shows the execution time for each of the 10 apps we test.

App Name	Execution Time	Lines of code in main activity
FooCam	0.2s	297
AntennaPod	1.5s	514
CoCoin	1.2s	770
LeafPic	1.6	633
Keepass2Android	1.1s	287
Camera2Basic	0.4s	1036
Location Samples	0.3s	241
OpenCamera	1.5	3038
Telegram	15s	4193
WordPress	14s	1546

Table 2: Execution time and Lines of code in the main activity for each app

Table 2 above depicts the execution time and lines of code for each of the 10 open source Android applications. The average execution time for the first 8 applications is 780 milliseconds. This shows that our tool can analyze, parse and check for resource acquisition and releasing for each of the 3 resources and for relatively large mobile apps in a short time (780 milliseconds).

The last two applications which are Telegram and WordPress are huge enterprise applications having hundreds of activities, and our tool takes about 14.5s on average to analyze such applications.

Chapter 7

Conclusion

Mobile application testing is of paramount importance. The overwhelming majority of professional Android application developers rely heavily on manual testing for the developed Android applications. The need for an automated testing solution approach has never been higher. In this research, we propose a model-based automated testing approach for testing Android applications automatically. We propose a model that focuses on the Android activity lifecycle callback methods. This model is then implemented into a proof-of-concept Java application tool that consumes a full Android application and generates a model of each activity, fragment and AppCompatActivity file in the form of a graph data structure. This graph contains the lifecycle methods and resources acquired and released as nodes, and the transitions between these lifecycle methods as edges. Then this graph is used for checking the acquisition/release of three Android system resources which are the camera, the location and the external drive (for read and write operations). This model simple has three outputs:

1. Text files that contain the lifecycle methods, their acquired and released resources and transitions between these lifecycle methods which mimic a graph data structure.
2. Application GUI status report which includes each resource (being one of the 3 tested resources), its parent activity, fragment or AppCompatActivity file name, and the status of that resource (pass/fail).
3. Inject *TODO* comment recommendations inside the Android applications source code to help the developer locate the resources that failed (acquired but not released), and in which lifecycle methods to release them.

In the end, we conduct an experiment on 10 open source Android applications. The results of this experiment seem promising and have good results. On average, for 8 out of the 10 applications we tested in this experiment, it only takes our tool around 780 milliseconds to parse and analyze the application and produce results. This indicates reasonable and good execution performance. Finally, we conduct a user acceptance test with 6 Android developer participants. The results of this test indicate that the automated testing application we propose is useful.

7.1 Threats to Validity

Even though we did our best to try and reduce threats to validity for both evaluation methods, we introduce the following threats to Validity.

7.1.1 User Evaluation Experiment

Regarding internal validity, our experiment's 6 participants were all software developers from the same software development company (Harri LLC). The affiliation between these developers may have influenced and biased their responses. Besides, some of the participants knew the researcher in person, so this might have affected their responses to the post experiment survey. When designing this user evaluation experiment, we had the novice mobile developers in mind as the target for this research. Even though the majority of our participants had less than one year of experience (more 66.7%), some had around 5 years of experience. This could have resulted in bias towards the questionnaire. That being said, novice developers can definitely benefit from this automated testing application.

When it comes to external threats to validity, we suspect some factors may have caused this type of threat. The user evaluation experiment involved 6 participants, even though this number

was sufficient for our purpose of measuring user acceptance, it may not be suitable for statistical analysis purposes. Evaluating our automated testing application with a wider range of participants and on a variety of mobile applications would provide us with more confidence in our tool. Besides, providing more applications for the users to test with and from different domains would help us expose gaps in our model and improve on it.

We also question our construct validity. The experiment asked the developers to modify the existing code to make sure the non released resources are released and in the right lifecycle methods. However, the experiment did not ask the developers to extend the features of the application to mimic actual software development scenarios, and to test acquired and released resources for a resource the developer himself/herserself implemented. Thus, not all steps of applications development and testing were evaluated by our experiment. Future work on our model-based testing application may include extending the sample application features and acquiring and releasing system resources, to make sure every part of the process is captured and evaluated.

7.1.1 Open Source Applications Evaluation

Our automated testing application was tested on 10 real world open source applications from multiple domains such as multimedia, communication and navigation. However, in order for us to have more confidence in the testing results, we need to test our automated testing application on more open source applications. Future work on this automated testing application may also cover applications from other platforms such as iOS applications.

7.2 Future Work

Our automated testing application involves checking for acquired and non released resources. Then we insert comments in the source code to recommend how and where to release the failed resources. Future work may include enhancing our suggested model to automated generated and insert the code to release the failed resources automatically. This is actually a feature that several developers in the survey had asked for. Besides, future work may include generalizing this model to cover mobile applications on other platforms such as iOS applications.

Future work may also cover integrating resource checks for the rest of the Android system resources. These resources include sensors, bluetooth, cellular data,...etc. Having all these resources integrated into the model and having sets of rules for each of these would definitely help generalize our automated testing application.

References

[1] H. Muccini, A. Di Francesco, and P. Esposito, “Software testing of mobile applications: Challenges and future research directions,” in *2012 7th International Workshop on Automation of Software Test (AST)*, Zurich, Switzerland, 2012, pp. 29–35.

[2] É. Payet and F. Spoto, “Static analysis of Android programs,” *Information and Software Technology*, vol. 54, no. 11, pp. 1192–1201, Nov. 2012.

[3] D. Franke, S. Kowalewski, C. Weise, and N. Prakobkosol, “Testing Conformance of Life Cycle Dependent Properties of Mobile Applications,” in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, Montreal, QC, Canada, 2012, pp. 241–250.

[4] S. Zein, N. Salleh, and J. Grundy, “Mobile Application Testing in Industrial Contexts: An Exploratory Multiple Case-Study,” in *Intelligent Software Methodologies, Tools and Techniques*, vol. 532, H. Fujita and G. Guizzi, Eds. Cham: Springer International Publishing, 2015, pp. 30–41.

[5] F. Nayebi, J.-M. Desharnais, and A. Abran, “The state of the art of mobile application usability evaluation,” in *2012 25th IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, Montreal, QC, 2012, pp. 1–4.

[6] R. Harrison, D. Flood, and D. Duce, “Usability of mobile applications: literature review and rationale for a new usability model,” *Journal of Interaction Science*, vol. 1, no. 1, p. 1, 2013.

[7] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, “Using GUI ripping for automated testing of Android applications,” in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*, Essen, Germany, 2012, p. 258.

[8] H. Kim, B. Choi, and W. E. Wong, “Performance Testing of Mobile Applications at the Unit Test Level,” in *2009 Third IEEE International Conference on Secure Software Integration and Reliability Improvement*, Shanghai, China, 2009, pp. 171–180.

[9] IDC: The premier global market intelligence company. (2020). IDC - Smartphone Market Share - OS. [online] Available at: https://www.idc.com/promo/smartphone-market-share/os [Accessed 18 Jan. 2020].

[10] D. Franke, S. Kowalewski, C. Weise, and N. Prakobkosol, “Testing Conformance of Life Cycle Dependent Properties of Mobile Applications,” in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, Montreal, QC, Canada, 2012, pp. 241–250, doi: 10.1109/ICST.2012.104.

[11] Android Developers. (2020). Understand the Activity Lifecycle | Android Developers. [online] Available at: https://developer.android.com/guide/components/activities/activity-lifecycle [Accessed 15 Jan. 2020].

[12] "Unit Testing - Software Testing Fundamentals", Software Testing Fundamentals, 2020. [Online]. Available: http://softwaretestingfundamentals.com/unit-testing/. [Accessed: 15- Jan- 2020].

[13] M. Linares-Vasquez, K. Moran, and D. Poshyvanyk, “Continuous, Evolutionary and Large-Scale: A New Perspective for Automated Mobile App Testing,” 2017, pp. 399–410, doi: 10.1109/ICSME.2017.27.

[14] D. Bernardo Silva, A. T. Endo, M. M. Eler and V. H. S. Durelli, "An analysis of automated tests for mobile Android applications," *2016 XLII Latin American Computing Conference (CLEI)*, Valparaiso, 2016, pp. 1-9.

[15] S. R. Choudhary, A. Gorla, and A. Orso, “Automated Test Input Generation for Android: Are We There Yet? (E),” in 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lincoln, NE, USA, 2015, pp. 429–440, doi: 10.1109/ASE.2015.89.

[16] “The Monkey UI android testing tool,” http://developer.android.com/ tools/help/monkey.html.

[17] A. Machiry, R. Tahiliani, and M. Naik, “Dynodroid: An Input Generation System for Android Apps,” in Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 224–234. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491450

[18] S. Anand, M. Naik, M. J. Harrold, and H. Yang, “Automated Concolic Testing of Smartphone Apps,” in Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, ser. FSE ’12. New York, NY, USA: ACM, 2012, pp. 59:1–59:11. [Online]. Available: http://doi.acm.org/10.1145/2393596.239366617 \

[19] T. Azim and I. Neamtiu, “Targeted and Depth-first Exploration for Systematic Testing of Android Apps,” in Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, ser. OOPSLA ’13. New York, NY, USA: ACM, 2013, pp. 641–660. [Online]. Available: http://doi.acm.org/10.1145/2509136.2509549

[20] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine and G. Imparato, "A toolset for GUI testing of Android applications," *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, Trento, 2012, pp. 650-653, doi: 10.1109/ICSM.2012.6405345

[21] G. de Cleva Farto and A. T. Endo, “Evaluating the Model-Based Testing Approach in the Context of Mobile Applications,” *Electronic Notes in Theoretical Computer Science*, vol. 314, pp. 3–21, Jun. 2015.

[22] Robotium, Robotium - the world's leading Android test automation framework. [Online; accessed 2014]. [Online]. Available: https://code.google.com/p/robotium

[23] N. H. Saad and N. S. Awang Abu Bakar, "Automated testing tools for mobile applications," *The 5th International Conference on Information and Communication Technology for The Muslim World (ICT4M)*, Kuching, 2014, pp. 1-5.

[24] K. Frajtak, M. Bures, and I. Jelinek, "Exploratory testing supported by automated reengineering of model of the system under test," *Cluster Computing*, vol. 20, no. 1, pp. 855–865, Mar. 2017.

[25] S. Subramanian, T. Singleton, and O. El Ariss, "Class coverage GUI testing for Android applications," in *2016 International Conference on System Reliability and Science (ICSRS)*, 2016, pp. 84–89.

[26] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "MobiGUITAR: Automated Model-Based Testing of Mobile Apps," *IEEE Software*, vol. 32, no. 5, pp. 53–59, Sep. 2015.

[27] A. R. Espada, M. del M. Gallardo, A. Salmerón, and P. Merino, "Using Model Checking to Generate Test Cases for Android Applications," *Electronic Proceedings in Theoretical Computer Science*, vol. 180, pp. 7–21, Apr. 2015.

[28] I.-A. Salihu, R. Ibrahim, B. S. Ahmed, K. Z. Zamli, and A. Usman, "AMOGA: A Static-Dynamic Model Generation Strategy for Mobile Apps Testing," *IEEE Access*, vol. 7, pp. 17158–17173, 2019.

[29] P. Liu, X. Zhang, M. Pistoia, Y. Zheng, M. Marques, and L. Zeng, "Automatic Text Input Generation for Mobile Testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, Buenos Aires, 2017, pp. 643–653.

[30] V. Gudmundsson, M. Lindvall, L. Aceto, J. Bergthorsson, and D. Ganesan, "Model-based Testing of Mobile Systems – An Empirical Study on QuizUp Android App," *Electronic Proceedings in Theoretical Computer Science*, vol. 208, pp. 16–30, May 2016.

[31] L. Panizo, A. Díaz, and B. García, "Model-based testing of apps in real network scenarios," *International Journal on Software Tools for Technology Transfer*, Apr. 2019.

[32] K. Frajták, M. Bures, and I. Jelinek, "Model-Based Testing and Exploratory Testing: Is Synergy Possible?," in *2016 6th International Conference on IT Convergence and Security (ICITCS)*, 2016, pp. 1–6.

[33] H. Zhang, H. Wu, and A. Rountev, "Automated test generation for detection of leaks in Android applications," in *Proceedings of the 11th International Workshop on Automation of Software Test - AST '16*, Austin, Texas, 2016, pp. 64–70.

[34] Y.-M. Baek and D.-H. Bae, "Automated model-based Android GUI testing using multi-level GUI comparison criteria," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*, Singapore, Singapore, 2016, pp. 238–249, doi: 10.1145/2970276.2970313.

[35] D. Amalfitano, N. Amatucci, A. R. Fasolino, and P. Tramontana, "A Conceptual Framework for the Comparison of Fully Automated GUI Testing Techniques," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, Lincoln, NE, 2015, pp. 50–57.

[36] S. Barnett, I. Avazpour, R. Vasa, and J. Grundy, "Supporting multi-view development for mobile applications," *Journal of Computer Languages*, vol. 51, pp. 88–96, Apr. 2019, doi: 10.1016/j.cola.2019.02.001.

[37] https://javaparser.org/

[38] N. Smith, “JavaParser: Visited,” p. 7.

[39] S. Zein, N. Salleh, and J. Grundy, “Static analysis of android apps for lifecycle conformance,” in *2017 8th International Conference on Information Technology (ICIT)*, Amman, Jordan, May 2017, pp. 102–109, doi: 10.1109/ICITECH.2017.8079982.

[40] *Payet, É. and F. Spoto, Static analysis of Android programs. [20] Information and Software Technology, 2012. 54(11): p. 1192-*

1201.

Appendix A: Survey Questions

Pre-experiment survey

1. How many years of experience do you have in Android app development?

- 1 or less
- 1 - 3 years
- 3 - 5 years
- 5 - 7 years
- More than 7

2. Gender?

- Male
- Female

3. How many Android apps have you built/helped build?

- 1 - 5
- 6 - 10
- more than 10

4. Have you used automated Android app testing tools before?

- yes
- no

5. What level of education do you have?

- High School
- Diploma
- Bachelor's Degree
- Master's Degree
- PhD

Post Experiment Survey

Answers to questions 1 through 15 have the following likert scale for answer

Questions 16 and 17 are open-ended questions

1. It was fairly easy to use the automated testing application

2. It was easy to learn how to use the automated testing application

3. It was easy to find the log files for each activity

4. It was easy to find the @TODO comments in the activity files for failing resources

5. It was easy to find the resources that were caught but not released in any activity lifecycle method

6. The results screen in the automated testing application GUI provided useful information about the Android app resources

7. The information in the automated testing application GUI helped me identify resources that are caught but not released

8. It was easy to release the non released (failed) resources and run the automated testing application again to verify if resource is released (passed)

9. It was easy to remember how to use the automated testing application again after releasing the resources in the Android app

10. It was easy to avoid making errors or mistakes while using the automated testing application

11. The automated testing application makes it easy to detect resources caught but not released

12. The automated testing application makes the Android app development and testing go faster

13. The automated testing application makes it more productive to develop and test Android apps

14. You are satisfied with using the automated testing application

15. You would recommend using this automated testing application to a fellow Android developer

16. What did you like the most about the automated testing application?

17. What features would you like to be added to the automated testing application?