Automated Support for Mobile Application Testing and Maintenance

Mattia Fazzini
Georgia Institute of Technology
Altanta, Georgia, USA
mfazzini@cc.gatech.edu

ABSTRACT

Mobile applications are an essential part of our daily life. In fact, they can be used for tasks that range from reading the news to performing bank transactions. Considering the impact that mobile applications have in our lives, it is important for developers to test them and gain confidence that they behave as expected. However, testing mobile applications proves to be challenging. In fact, mobile companies report that they do not have enough time and the right methods to test. In addition, in the case of Android applications, the situation is further complicated by the "fragmentation" of the ecosystem. Developers not only need to ensure that an application behaves as expected but also need to make sure that the application does so on a multitude of different devices. Finally, because it is virtually impossible to release a bug free application, developers also need to quickly react to bug reports and release a fixed version of the application before customer loss. The research plan proposed in this paper, aims to provide novel techniques to automate the support for mobile application testing and maintenance. Specifically, it proposes techniques to: test apps more effectively and efficiently, tackle the problems caused by the "fragmentation" of the Android ecosystem, and help developers in quickly handling bug reports.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

Mobile apps, automated testing, differential testing, debugging

ACM Reference Format:
Mattia Fazzini. 2018. Automated Support for Mobile Application Testing and Maintenance. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3236024.3275425

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3275425

1 INTRODUCTION

Mobile devices [1] are becoming the prevalent form of computation and the most popular way of accessing digital media content [6]. Mobile applications (or simply apps) perform an essential role in the success story of mobile devices and have fundamentally impacted our lives. In fact, apps can be used to facilitate many of our daily activities, such as shopping, banking, social networking, and traveling. It is therefore not a surprise that apps result to be the type of software that is principally used on mobile devices [15].

Apps, similarly to all other software applications, must be tested to gain confidence that they behave as expected. This is especially important in the context of mobile apps where they are part of a highly competitive market and a failure in an app can result in a loss of reputation and ultimately customers. In fact, an astonishing 88% of app users would consider abandoning an app if they encountered bugs or glitches [3]. At the same time, app testing shows to be increasingly challenging. A study [4] involving 1660 companies from 32 different countries reports that: 52% of them do not have enough time to test, 47% of them do not have the right testing process or method, 46% of them do not have the right tools to test, 42% of them do not have mobile testing experts, 41% of them do not have in-house testing environment, and 40% of them do not have testing devices readily available. Therefore is necessary for the software engineering community to investigate and propose novel mobile app testing techniques in order to overcome the current challenges faced by companies and their developers.

In the case of Android apps, the task of establishing whether an app behaves as expected is further complicated by the "fragmentation" of the ecosystem. Given the open source nature of the Android system and vendors' intent to satisfy different market needs, Android devices are often customized (in terms of both their hardware and software) leading to a multitude of devices concurrently available in the field. For instance, the number of distinct devices in the year 2015 was 24,093 [19]. Different device properties can distinctively affect the execution of an app, leading to compatibility issues for the app on some devices. There can be different types of compatibility issues. First, different values in the size, the resolution, or the pixel density of screens can affect how the app is displayed. This characteristic can lead to have an app that is not properly displayed on some devices. Second, apps execute on devices that can run different versions or customizations of the operating system. Because certain system APIs can execute differently across versions or customizations of the system, apps using such APIs can receive unexpected inputs from the system leading to issues in their execution. Third, because devices come

[1] The term device is used to refer both to its hardware and operating system.

from different manufactures, they tend to have different hardware components, which can lead different devices to have different capabilities. For this reason, apps might not execute as expected on devices where a certain hardware component is missing. Given the "fragmentation" of the Android ecosystem, a developer of an Android app not only needs to gain confidence that an app behaves as expected but also needs to determine whether the app does so across different devices. For this reason, new techniques that focus on the "fragmentation" of the ecosystem are required to provide the necessary confidence to developers that apps behave as intended.

At the same time, because app development is driven by demanding time to market constraints [21] and because software verification techniques have inherent limitations, it is virtually impossible for developers to identify all bugs before releasing apps. Consequently, it is possible for users to experience failures. In order to quickly address such failures, companies provide automated or manual mechanisms that users can utilize to send bug reports directly to developers. Bug reports contain information describing a failure and developers analyze them to try to understand and fix the bug causing a failure. Considering that the number of bug reports can be significantly high and that bug reports come with possibly incomplete information [1, 2], these tasks show to be time consuming and expensive [22]. A particular aspect of mobile apps, is the availability of a high number of natural language bug reports [17, 20]. Having automated techniques that are able to analyze such bug reports and help developers in task of fixing bugs, would allow developers to quickly release a fixed version of the app and avoid possible loss of reputation and users.

The objective of the research proposed in this paper is to design novel techniques to address the challenges presented above.

2 PRELIMINARY RESEARCH

The research proposed in this paper is inspired by promising results from preliminary research. First, we developed an app testing technique [7] that generates device independent test cases so that developers can be more effective and efficient in testing apps. Second, we designed a technique [8] that identifies screen compatibility issues caused by the "fragmentation" of the ecosystem so that such issues can be addressed before app release. Finally, we proposed a technique [9] that automatically translates bug reports into test cases so that developers can promptly start repairing bugs.

2.1 Test Case Generation

In the first part of our preliminary research, we designed a technique that records, encodes, and runs device independent test cases for Android apps [7]. The technique is characterized by the following aspects. First, the technique implements the record once-run everywhere principle. In fact, developers can record their tests on one device and rerun them on other devices. Second, the technique allows developers to create oracles in an intuitive way by interacting with the GUI of the app under test (AUT). Finally, the technique is minimally intrusive as it does not require any modification to the device on which the AUT is running.

The technique consists of three main phases. In the *test case recording phase*, the developer interacts with the AUT with the objective of testing its functionality. In this phase, the technique records user interactions and offers an interface to define assertion-based oracles. The technique is able to record interactions and oracles by leveraging the accessibility subsystem of the Android framework. The technique encodes interactions and oracles into a trace. For interactions, the technique encodes: the type of interaction, the GUI element exercised by the interaction, and the properties of the interaction. For oracles, the technique encodes: the type of oracle, the GUI element checked by the oracle, and the properties of the oracle. To execute interactions and check oracles on devices different from the one in which they were defined, the technique uses the concept of *selector*. A selector uniquely identifies an element in the GUI and is independent from the size, the resolution, and the pixel density of the screen. A selector is computed by analyzing: identifiers, XPaths, and properties of GUI elements. At the end of the recording phase, the technique enters its *test case encoding phase*, which extracts recorded interactions and oracles from the trace and translates them into test cases. Finally, in the *test case execution phase*, the technique executes test cases on multiple devices and summarizes test results in a report.

2.1.1 Results. We implemented the technique in a tool and performed a user study to evaluate expressiveness, efficiency, and ultimately usefulness of the technique. The study compares the technique to a baseline made by two other test case recording techniques: Testdroid Recorder (TR) [14] and Espresso Test Recorder (ETR) [10]. Based on the results of the study, we conclude that the technique can record test cases, and is more expressive and efficient than the baseline. Furthermore, test cases recorded using the technique can run on different devices in the majority of cases.

2.2 Compatibility Issue Analysis

In the second part of our preliminary research, we developed a technique based on differential testing that automatically identifies compatibility issues generated by the "fragmentation" of the Android ecosystem [8]. The technique finds issues caused by the "fragmentation" in the size, resolution, and density of device screens.

The technique is composed by four main phases and takes as inputs: the AUT, a reference device, and a set of test devices. In its first phase, the technique dynamically generates inputs on the reference device with the objective of testing the app's functionality. Before exercising the app with an input, the technique encodes the current state of the GUI into a model called the *window model*. The model is composed of a tree abstraction and a screenshot of the GUI. The technique logs inputs and window models into a trace, which is the input to the following phase. In its second phase, the technique creates a *UI model* for the reference device by extracting window models contained in the trace and translates inputs from the trace into a device independent test case. The test case also captures window models before providing test inputs to the AUT. The third phase runs the test case on the set of test devices and creates a UI model for each device. The fourth phase identifies compatibility issues by comparing the UI model of the reference device with the UI model of test devices. Specifically, the technique identifies two types of compatibility issues: *structural compatibility issues* and *visual compatibility issues*. Structural compatibility issues are identified by comparing a GUI tree from the reference device with the corresponding GUI tree from a test device. The

comparison is based on a node matching algorithm. The technique identifies visual compatibility issues by comparing the visual representation of a GUI element in the reference device with the visual representation of the corresponding GUI element in a test device. The comparison is performed using a decision tree classifier. Finally, identified compatibility issues are reported to the developer.

2.2.1 Results. We evaluated the technique using five real-world apps and 147 different devices. Across all apps and devices, the technique identified six structural compatibility issues and 89 visual compatibility issues while reporting 16 false positives. The technique is able to analyze a single app overnight. By manually analyzing issues reported, we found that the devices that are most problematic are the ones with small screen size and low resolution. However, simply analyzing such devices would have not allowed the technique to identify all issues reported.

2.3 Bug Reports Analysis

In the third part of our preliminary research, we devised a technique to automatically translate bug reports into test cases [9]. The technique focuses on natural language reports describing how to reproduce a bug in terms of GUI actions. The test case generated by the technique can be used by a developer to debug and fix the app.

The technique is characterized by three main phases: the *ontology extraction phase*, the *bug report analysis phase*, and the *GUI actions search phase*. In the first phase, the technique takes as input the *relevant app* associated with the report, and extracts a description of GUI elements used in the app. This description is stored in the *ontology* for the app and is used by the second phase of the technique to create a mapping between the vocabulary used in the bug report and the GUI elements of the relevant app. The second phase of the technique analyzes the bug report to extract a list of *abstract steps*. Abstract steps encode actions described in the bug report. In this phase, the technique analyzes clauses in the report using their dependency tree representation [5, 13]. For each clause, the technique extracts the action to be performed on the GUI, the target GUI element of the action, and the properties of the action. The technique recognizes a GUI element in the description of the report by comparing the text of the report with the content of the ontology using an approach based on word embeddings. The third phase of the technique dynamically explores the relevant app looking for a sequence of GUI actions that match abstract steps. The exploration uses backtracking to account for the fact that there could be multiple candidate GUI elements as the target of an abstract step. The exploration also uses random input generation to account for the fact that a report might not list all steps necessary to reproduce the bug. The final output of the technique is a test case that can be used to debug and fix the app.

2.3.1 Results. To evaluate the effectiveness of the technique, we applied its implementation to a set of randomly selected bug reports. The technique was able to translate 59.7% of the reports (19.4% of the reports was trivial to translate as it required to simply open the app). Among these reports, the technique was also able to translate bug reports that had missing steps necessary to reproduce the bug. Finally, when successful, the technique could translate any single report in less than 30 minutes. This execution time suggests that the technique could be used to monitor bug reports and generate test cases throughout the day, as opposed to overnight only.

3 PROPOSED RESEARCH

Our preliminary research made initial steps toward addressing the challenges presented in Section 1. The positive results encourage us to extend our preliminary research into a research program that we detail in this section.

3.1 Test Case Generation

In the first part of our research plan, we will extend different aspects of our work on test case generation [7].

First, we plan to add sandboxing capabilities to our record and replay technique. In this way, developers can run recorded test cases independently from external dependencies. Sandboxing is particularly important in the context of mobile devices as certain bugs manifest only under specific conditions. For instance, a bug can appear when the user of an app moves around and the connection type of the device changes. To develop sandboxing capabilities, we will take advantage of the insights from related work [12] and create a technique that does not require any modification to the underlying operating system. This characteristic is important for the adoption of the technique considering the number of devices present in the field [19]. Second, we plan to use fuzzing, guided by developer defined coverage goals, for generating extra tests starting from the ones already recorded. Finally, we plan to create a version of our record and replay technique that can be used to perform on-device bug reporting. With such a technique, users could report executable test cases reproducing the bug directly to developers.

3.2 Compatibility Issue Analysis

In the second part of our research plan, we will extend our work [8] on the "fragmentation" of the Android ecosystem to consider additional types of compatibility issues. We will also design techniques to automatically repair apps that exhibit compatibility issues.

First, we plan to investigate compatibility issues generated by different versions and customizations of the Android operating system. Specifically, we plan to analyze how the behavior of operating system APIs from different devices can affect the execution of Android apps. Related work [23] manually analyzes a set of reported issues and proposes a static analysis to detect such issues. In our work, we plan to have a more general technique based on differential testing that identifies differences in the expected behavior of an API and reports them to developers. Specifically, we will investigate ways to automatically exercise APIs used by an app under different inputs and conditions, encode the answer from APIs into a model, and compare the model generated from a reference device with the models extracted from a set of test devices. Second, we plan to design a technique that is able to identify whether an app behaves as expected when executing on a device that does not offer an hardware component used by the app. Finally, we plan to develop a technique that is able to automatically update outdated or deprecated APIs. Related work [11, 16] is able to automatically identify when an app is using an API that would generate an issue when invoked on a certain version of the operating system. In our

work, we plan to automatically repair such API usages by analyzing examples from other apps that have already fixed such API usages.

3.3 Bug Reports Analysis

In the third part of our research plan, we will extend our work on the analysis of bug reports [9]. Specifically, we will focus our attention on helping users to write better reports and assist developers in quickly triaging bug reports.

First, we plan to design a technique that helps users in writing natural language bug reports and at the same time automatically generates a test case for the report. On the user side, the technique will work as a content assist and will suggest actions to be performed on GUI elements while the user is typing the report. Actions and GUI elements are suggested to the user by running the app affected by the bug in the background, replicating the actions provided by the user, and analyzing the information displayed by the GUI. This technique differs from related work [18] as it is able to explore the app while the user is typing the report. This characteristic can lead to a high accuracy for generated test cases. Second, we plan to develop a technique able to help developers in triaging bug reports. The technique will translate bug reports written in natural language into test cases and then perform differential testing to observe whether two test cases lead to the same crash. If this is the case, the technique will report them as duplicate bug reports. The same technique can also be used to prioritize bug reports by looking at coverage information for all bug reports. Finally, we plan to extend our technique that translates bug reports into test cases to handle operations that are semantically related to the functionality of the app but do not explicitly mention GUI actions. (e.g., "browse through the folders of the app"). This technique will leverage an existing test suite associated with an app and map test input sequences to semantic operations by looking at the name of app methods exercised by the inputs.

4 EXPECTED CONTRIBUTIONS

The research plan proposed in this paper aims to provide novel techniques and tools for testing and maintaining mobile apps. The techniques will help developers in testing apps more effectively and efficiently, give developers mechanisms to tackle the "fragmentation" of the Android ecosystem, and quickly handle bug reports. We foresee our research to have an impact both in the research and industry communities. In fact, we plan to open source the implementation of the techniques (similarly to what we have done in preliminary research) to foster additional contributions from the research community. Second, we plan to directly involve developers into the evaluation of the techniques to gather feedback from real world scenarios. Finally, we will help developers to integrate our techniques in their development processes.

ACKNOWLEDGMENTS

The author is advised by Dr. Alessandro Orso. The research work described in this paper was partially supported by the NSF under grants CCF-1453474, CCF-1564162, CCF-1320783, CCF-1161821, and CCF-1563991, and by funding from Amazon, Google, IBM Research, and Microsoft Research.

REFERENCES

[1] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. 2008. What Makes a Good Bug Report?. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, New York, NY, USA, 308–318.

[2] Pamela Bhattacharya, Liudmila Ulanova, Iulian Neamtiu, and Sai Charan Koduru. 2013. An Empirical Analysis of Bug Reports and Bug Fixing in Open Source Android Apps. In *17th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, Washington, DC, USA, 133–143.

[3] David Bolton. 2017. 88% Of People Will Abandon An App Because Of Bugs. Retrieved June 29, 2018 from https://www.applause.com/blog/app-abandonment-bug-testing

[4] Capgemini 2018. World Quality Report 2017-18. Retrieved June 29, 2018 from https://www.capgemini.com/service/world-quality-report-2017-18

[5] Marie-Catherine de Marneffe, Bill MacCartney, and Christopher D. Manning. 2006. Generating Typed Dependency Parses from Phrase Structure Parses. In *Proceedings of the Fifth International Conference on Language Resources and Evaluation*. European Language Resources Association (ELRA), Genoa, Italy, 449–454.

[6] Darrell Etherington. 2016. Mobile internet use passes desktop for the first time, study finds. Retrieved June 29, 2018 from https://techcrunch.com/2016/11/01/mobile-internet-use-passes-desktop-for-the-first-time-study-finds

[7] Mattia Fazzini, Eduardo Noronha de A. Freitas, Shauvik Roy Choudhary, and Alessandro Orso. 2017. Barista: A Technique for Recording, Encoding, and Running Platform Independent Android Tests. In *2017 IEEE International Conference on Software Testing, Verification and Validation*. IEEE Computer Society, Washington, DC, USA, 149–160.

[8] Mattia Fazzini and Alessandro Orso. 2017. Automated Cross-Platform Inconsistency Detection for Mobile Apps. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, Washington, DC, USA, 308–318.

[9] Mattia Fazzini, Martin Prammer, and Marcelo d'Amorim Alessandro Orso. 2018. Automatically Translating Bug Reports into Test Cases for Mobile Apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, New York, NY, USA, 141–152.

[10] Google 2016. Create UI tests with Espresso Test Recorder. Retrieved June 29, 2018 from https://developer.android.com/studio/test/espresso-test-recorder

[11] Dongjie He, Lian Li, Lei Wang, Hengjie Zheng, Guangwei Li, and Jingling Xue. 2018. Understanding and Detecting Evolution-induced Compatibility Issues in Android Apps. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, New York, NY, USA, 167–177.

[12] Yongjian Hu, Tanzirul Azim, and Iulian Neamtiu. 2015. Versatile yet Lightweight Record-and-Replay for Android. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, New York, NY, USA, 349–366.

[13] Dan Jurafsky and James H Martin. 2014. *Speech and language processing*. Pearson Education, London, UK.

[14] Jouko Kaasila, Denzil Ferreira, Vassilis Kostakos, and Timo Ojala. 2012. Testdroid: automated remote UI testing on Android. In *11th International Conference on Mobile and Ubiquitous Multimedia*. ACM, New York, NY, USA, 28–31.

[15] Adam Lella and Andrew Lipsman. 2017. The 2017 U.S. Mobile App Report. Retrieved June 29, 2018 from https://www.comscore.com/Insights/Presentations-and-Whitepapers/2017/The-2017-US-Mobile-App-Report

[16] Li Li, Tegawendé F. Bissyandé, Haoyu Wang, and Jacques Klein. 2018. CiD: Automating the Detection of API-related Compatibility Issues in Android Apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, New York, NY, USA, 153–163.

[17] Walid Maalej and Hadeer Nabil. 2015. Bug Report, Feature Request, or Simply Praise? On Automatically Classifying App Reviews. In *International Requirements Engineering Conference*. IEEE Computer Society, Washington, DC, USA, 116–125.

[18] Kevin Moran, Mario Linares Vásquez, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2015. Auto-Completing Bug Reports for Android Applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, New York, NY, USA, 673–686.

[19] OpenSignal 2015. Android Fragmentation. Retrieved June 29, 2018 from https://opensignal.com/reports/2015/08/android-fragmentation

[20] Dennis Pagano and Walid Maalej. 2013. User Feedback in the AppStore: An Empirical Study. In *21st IEEE International Requirements Engineering Conference*. IEEE Computer Society, Washington, DC, USA, 125–134.

[21] Avinash Sharma. 2018. 8 Quick Tips to Speed Up Android App Development. Retrieved June 29, 2018 from https://appinventiv.com/blog/8-quick-tips-speed-android-app-development

[22] Gregory Tassey. 2002. The Economic Impacts of Inadequate Infrastructure for Software Testing. *National Institute of Standards and Technology* 7007.011 (2002).

[23] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2016. Taming Android Fragmentation: Characterizing and Detecting Compatibility Issues for Android Apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, New York, NY, USA, 226–237.