Regression Testing of Android Applications

Quan Do
Advisor: Dr. Guowei Yang
Department of Computer Science
Texas State University
q_d2@txstate.edu
gyang@txstate.edu

ABSTRACT

As the mobile platform pervades human life, much research in recent years has focused on improving the reliability of mobile applications on this platform, for example by applying automatic testing. To date, however, researchers have primarily considered testing of single version of mobile applications. It has been shown that testing of mobile applications can be expensive; thus simply re-executing all tests on the modified application version presents challenges. Regression testing---a process of validating modified software to ensure that changes are correct and do not adversely affect other features of the software---has been extensively studied for desktop application, and many efficient and effective approaches have been proposed; however, these approaches cannot be directly applied to mobile applications. In this study, we propose a regression testing approach for mobile applications. Our approach leverages the combination of static impact analysis with code coverage that is dynamically generated at run-time, and identify a subset of tests to check the behaviors of the modified version that can potentially be different from the original version. We implement our approach for Google Android applications, and demonstrate its effectiveness using a case study.

1. INTRODUCTION

Mobile devices have become ubiquitous in modern society. The mobile platform is separating itself from a variety of areas of desktop applications such as entertainment, e-commerce and social media. Thus, developers are required to produce high quality mobile apps in terms of portability, reliability and security. In recent years, a large number of research projects has focused on improving the reliability of mobile applications on mobile platform, for example by applying automatic testing. However, the majority of the research are only focusing on testing of single version of mobile applications. It has been shown that testing of mobile applications is not trivial task and can be expensive; thus simply re-executing all tests on the modified application version presents challenges. As developers maintain a

software system, they periodically regression test it, to find errors that caused by their changes and provide confidence that their modifications are correct. To support this process, developers often create an initial test suite, and then reuse for regression testing [1]. Regression testing---a process of validating modified software to ensure that changes are correct and do not adversely affect other features of the software---has been extensively studied for desktop applications, and many efficient and effective approaches have been proposed; however, these approaches cannot be directly applied to mobile applications. One of the factors that causes the incompatibility is the different between the mobile platform's system architectures and the desktop platforms'. For example, although Android Platform [2] is written in Java as other Java desktop applications, it runs under the Dalvik Virtual Machine (DVM). The DVM and the Java byte-code run-time environment are substantially different [3]. In this study, we propose a regression testing approach for mobile applications, specifically the Android platform. Our approach leverages the combination of static impact analysis with code coverage that is dynamically generated at run-time, and identify a subset of tests to check the behaviors of the modified version that can potentially be different from the original version. We implement our approach for Google Android applications, and demonstrate its effectiveness using a case study.

2. MOTIVATING EXAMPLE

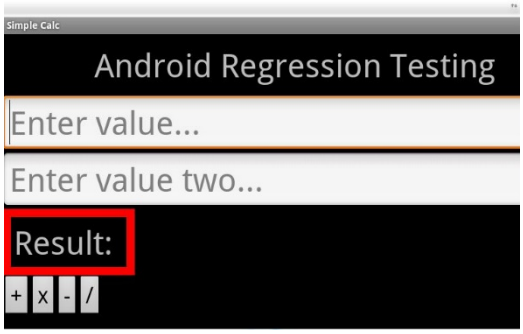

Figure 1: A screenshot of SimpleCalc Application

Figure 1 is a screenshot of *Simple Calculator Application,* a basic calculator that inputs two numbers and performs different arithmetic including addition, subtraction, multiplication and division by clicking

buttons *"+", "-", "x",* and *"/"* respectively. After debugging the app, two bugs are found at line number 4 and 22. In term of maintenance, after modifying an implementation, software is required to be re-tested in order to assure that changes do not adversely affect other software components. In the traditional regression testing techniques, first, a test suite is executed by the original program to produce test oracle. Then, the entire test suite is rerun against the modified code to produce test results. Finally, the results are compared against the test oracle to determine whether the changes affect features that normally operated prior to the modifications. In the Android Platform, these approaches remain challenge due to the cost of executing a large number of test cases. Therefore, reducing the number of test cases is significantly important. Instead of rerunning the entire test suite, our approach eliminated the test cases that are affected by the modifications.

Let *T* be a test suite for arithmetic testing.

$T = \{T_1, T_2, \ldots, T_{21}\}$ with

T_1: assertEquals("16", add("12"+"4"))

T_2: assertEquals("10.5", add("5.5"+"5"))

T_3: assertEquals("-9", add("3"+"-12"))

T_4: assertEquals("4", subtract("10"-"6"))

T_5: assertEquals("-4", subtract("2"-"6"))

T_6: assertEquals("-14", subtract("-10"-"4"))

T_7: assertEquals("2.4", subtract("8.8"-"6.4"))

T_8: assertEquals("0", multiply("0"*"7"))

T_9: assertEquals("-1", multiply("1"*"-1"))

T_{10}: assertEquals("77175", multiply("245"*"315"))

T_{11}: assertEquals("77175", multiply("315"*"245"))

T_{12}: assertEquals("-253", multiply("+11"*"+23"))

T_{13}: assertEquals("-253", multiply("-11"*"+23"))

T_{14}: assertEquals("4", divide("12"/"3"))

T_{15}: assertEquals("2.4", divide("4.8"/"2.4"))

T_{16}: assertEquals("0", divide("0"/"1"))

T_{17}: assertEquals("2", divide("2"/"1"))

T_{18}: assertEquals("0.2", divide("1"/"5"))

T_{19}: assertEquals("-4", divide("-12"/"3"))

T_{20}: assertEquals("4", divide("-12"/"-3"))

T_{21}: assertEquals("-4", divide("12"/"-3"))

After executing the test suite, a code coverage report for each test case is generated as described in Figure 2. For example, code coverage report for T_1, T_2, T_3 is 1->6, which indicates the source code line numbers 1 to 6.

Let P^O be the original program. P^O is modified at line number 4 (x*y becomes x+y) and at line number 22 (x-y becomes x/y). Let P^M be the modified program. After the modifications, it is required that only a subset, *T'*, of *T* is selected for re-execution. From the code coverage reports, let T~ be a set of affected test case, it is easy to show that $T\sim=\{T_1, T_2, T_3, T_{14}, T_{15}, T_{16}, T_{17}, T_{18}, T_{19}, T_{20}, T_{21}\}$ is not eligible for re-execution. Only a subset of T, that is, $T'=\{T_4, T_5, T_6, T_7, T_8, T_9, T_{10}, T_{11}, T_{12}, T_{13}\}$ is selected for re-execution. As the result, only 10 test cases are selected. Therefore, almost 50% of the number of test case are reduced.

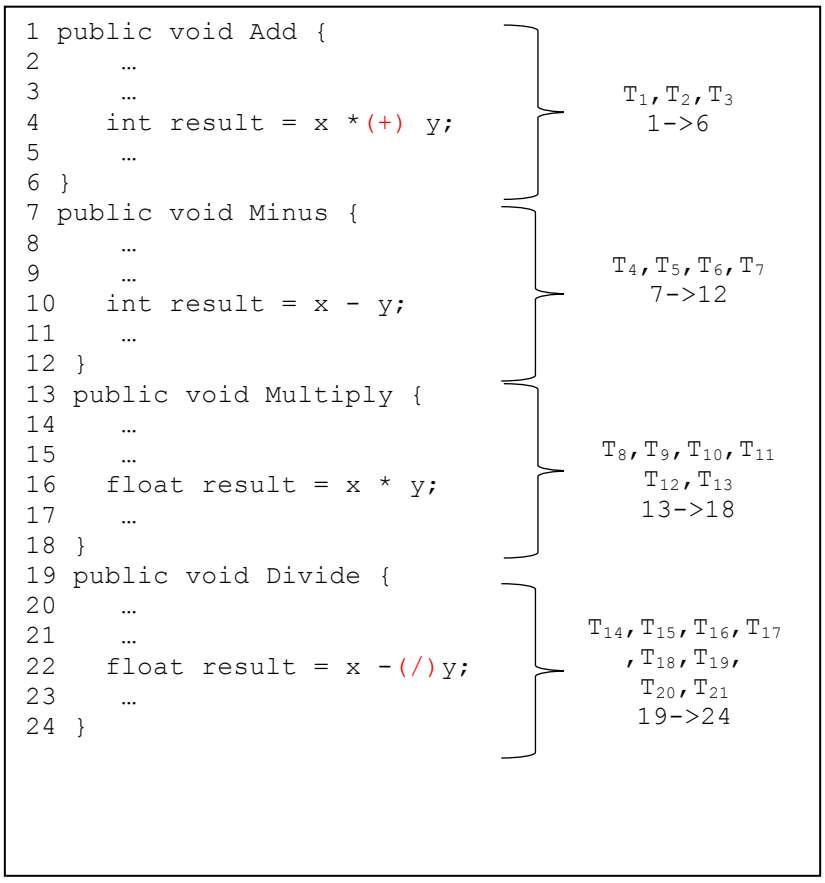

Figure 2: Implementations of four arithmetic methods

3. BACKGROUND

Android Development Platform

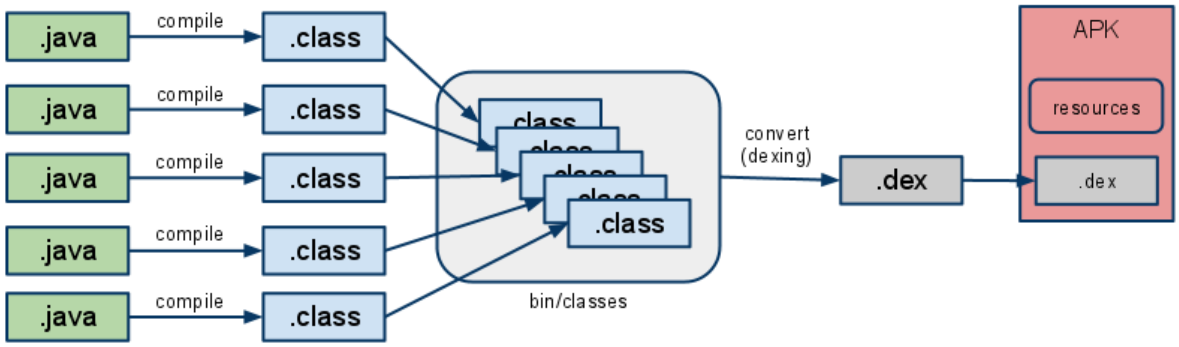

Figure 3: Android Development Architecture

Each Android application is executed under the Davik Virtual Machine (DVM) and Android applications are written in Java programming language. As described in Figure 3, first, the java source code `.java` files are compiled to `.class` bytecode files using standard Java Virtual Machine (JVM). After the compilation, the Davik `dx` tool converts the `.class` files to Davik bytecode files, and composes them into one single `.dex` file. The `.dex` file contains all applications classes. Finally, in order to run on an Android device, the `.dex` file and resources are composed into one executable `.apk` file and the Android device proceeds installation using the `.apk` file. Our approach assumes that application's source code is always available.

Regression Testing. Our approach leverages a regression test selection technique that has been used in our previous work [4]. Let P be a program, let P' be a modified version of P, and let T be a test suite developed for P. Regression testing is concerned with validating P'. Reusing all of T can be expensive, so regression test selection (RTS) techniques (see [11]) use data on P, P', and T to select a subset T' of T

with which to test P'. One class of RTS techniques, safe techniques, (e.g. [12]) guarantee that under certain conditions, test case not selected could not have exposed faults in P' [11]. Empirical studies have shown that these techniques can be cost-effective. In this work we construct control-flow graph (CFG) representations of the procedures in P and P', in which individual nodes are labeled by their corresponding statements. We perform a depth-first graph walk on a pair of CFGs G and G' for each procedure and its modified version in P and P', following identically-labeled edges, to find code changes. Given two edges e and e' in G and G', if the code associated with nodes reached by e and e' differs, we call e a dangerous edge: it leads to code that may cause program executions to exhibit different behavior.

5. APPROACH

Our approach leverages the combination of static impact analysis with code coverage that is dynamically generated at run-time. We separate the implementation into three components: code coverage generator, impact analyzer, and test case selector. First, we apply Emma [6], an open-source tool for measuring and reporting Java code coverage, on the original program to generate a code coverage report for each test case.

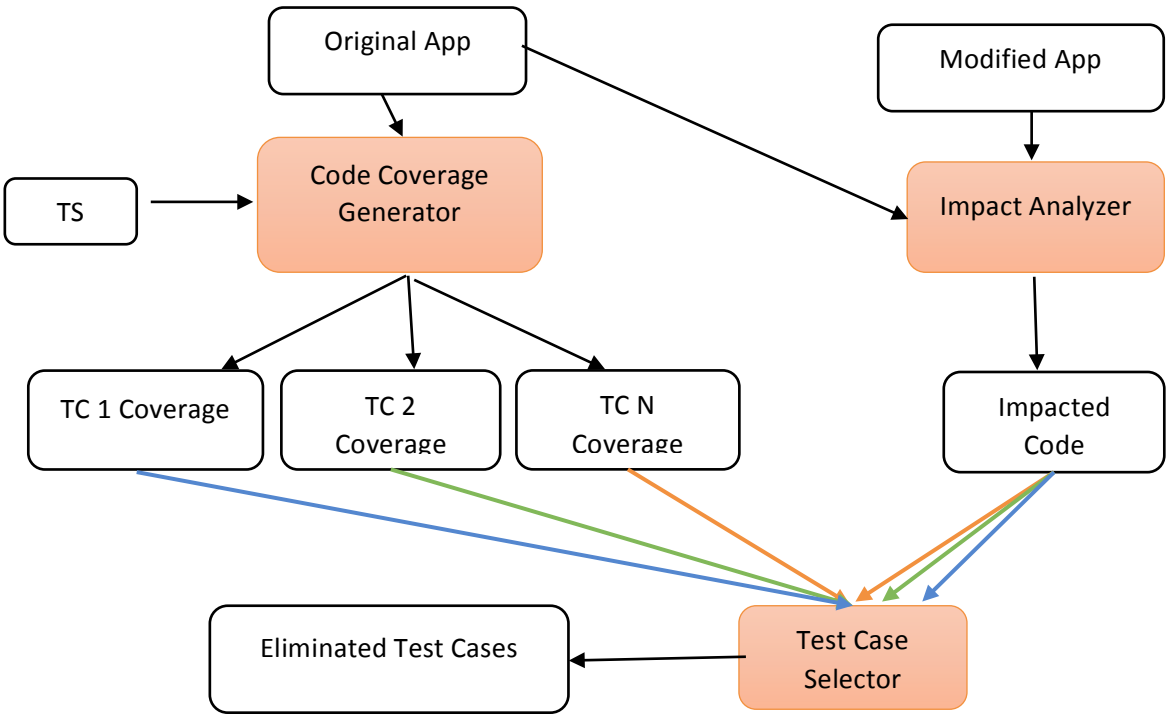

Figure 4: Approach Architecture

A

repo

rt contains source code line numbers that are covered after executing a test case. Next, the impact analyzer is implemented by using BCEL [5]. First, a Control Flow Graph (CFG) is built for each version of the application. The CFGs contain a set of nodes that represents bytecode statements and set of edges that represent the flow of control (branches) among statements. The impact analyzer then inputs the original program's and the modified program's bytecode and produces a report that contains the exact locations (source code line numbers) where modifications occur.

Each test case report is compared against the impacting report to determine which test case should be eliminated.

6. EVALUATION

In order to evaluate our approach, we use the above motivated example as a case study to verify the effectiveness of the test case selection technique. From Figure 5, in the methods `clickAddButton()` and `clickMultiplyButton()`, at line number 99, "+" sign is modified to "/" sign, and at line number 110, "*" is modified to "-" sign. After running the impact analyzer, a file named `impact` is

```
94      public void  clickAddButton(View view){
95          try {
96              int val1 = Integer.parseInt(value1.getText().toString());
97              int val2 = Integer.parseInt(value2.getText().toString());
98
99              Integer answer = val1 / val2;
100             result.setText(answer.toString());
101         } catch (Exception e) {
102             Log.e(LOG_TAG, "Failed to add numbers", e);
103         }
104     }
105     public void  clickMultiplyButton(View view){
106         try {
107             int val1 = Integer.parseInt(value1.getText().toString());
108             int val2 = Integer.parseInt(value2.getText().toString());
109
110             Integer answer = val1 - val2;
111             result.setText(answer.toString());
112         } catch (Exception e) {
113             Log.e(LOG_TAG, "Failed to multiply numbers", e);
114         }
115     }
```

generated containing the modified source code line numbers, which are 99 and 110.

Figure 5: Actual implementations of Addition and Multiplication methods

Then each test case is executed and its corresponding code coverage report is generated. Table 1 shows details of the code coverage report for each test case.

Comparing each test case report against the `impact` file, it clearly shows that the `add` test case and `multiply` test case are eliminated since they contain the impacted line numbers. Figure 6 shows the results of eliminated test cases.

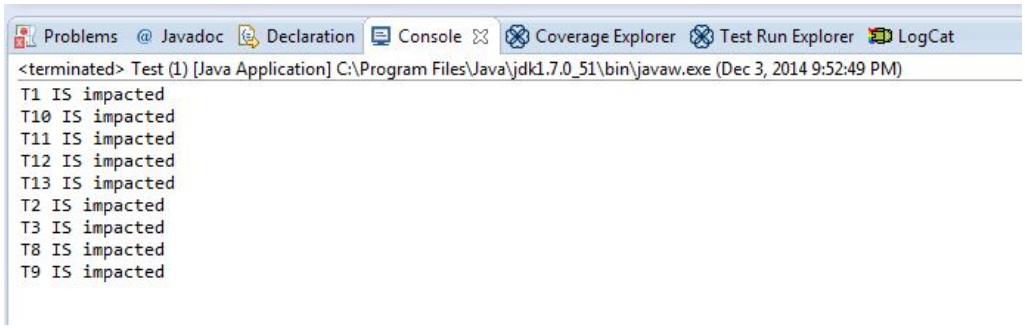

Figure 6: Eliminated Test Cases

7. RELATED WORK

Many researches have been conducted only focusing on testing a single version of application. For example, Azim et al. [13] presents an approach that allows popular Android apps to be explored systematically while running on actual phone. This work successfully addressed its purpose in terms of GUI exploration. Since the work is done without accessing to source code, but by the analysis of bytecode, all possible transitions (i.e. methods invocations) to new activities should be taken into account. Another single version application testing example is the work of Wei Yang et al. [10]. The researchers proposed a tool that implements the grey-box approach of automated model extraction for Android apps and its evaluation in demonstrating the effectiveness at generating high-quality GUI models.

Amalfitano et al. [7] proposed a type of regression testing of Android applications using the Monkey Runner tool [8] to check the modifications by comparing the output screenshots to a set of screenshots that are known to be correct. This approach only provides a high level of comparison to find the impact.

Crussell et al. [14] detects the similarity between an Android application's versions by constructing Program Dependency Graph (PDG) for each version using an existing tool and comparing the PDGs to find ***semantically*** code at the method level. Compared to our approach, we construct the CFGs (similar to PDGs) from scratch, which is more flexible to manipulate bytecode. Then we find ***syntactically*** code changed, that is, source code line number

Memon et al, [9] proposed a regression testing technique for GUIs application in general. The study only focuses on desktop applications instead of mobile applications.

8. CONCLUSION

We presented a new approach for regression testing of mobile platform. The approach leverages the concept of static analysis to detect the modifications on different versions of an Android application. Due the changes, only a subset of test cases is selected to avoid the cost of executing the entire test suite. We have implemented and evaluated our approach. The result has shown the effectiveness of test case selection.

We plan to improve some limitations of this approach in our future work. Instead of having access to the application's source code, we will apply the same regression testing technique on Android platform using only the `.apk` files. Comparing to this approach, the future approach will be required some conversions and mappings between `.dex` files and source code, `.dex` files and normal Java `bytecode`. In addition, not only do we focus on the physical modifications of the sources code, but also the changes in behavior of the application.

REFERENCES

[1] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, Gregg Rothermel. An Empirical Study of Regression Test Selection Techniques, *ACM Transactions on Software Engineering and Methodology, Vol. 10, No. 2, April 2001*

[2] Android Developers. The Developer's Guide. Available at: http://developer.android.com/

[3] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A Study of Android Application Security. *Proceeding SEC'11 Proceedings of the 20th USENIX coference on Security. Pages 21-21*.

[4] Guowei Yang, Matthew B. Dwyer, Gregg Rothermel. Regression Model Checking. *25th IEEE International Conference on Software Maintenance (ICSM 2009), pages 115--124, Edmonton, Elberta, Canada, September 2009.*

[5] http://jakarta.apache.org/bcel

[6] Google Code. Maven-android-plugin. https://code.google.com/p/maven-android-plugin/

[7] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana. A GUI Crawling-based technique for Android Mobile Application Testing. *I CSTW '11 Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops. Pages 252-261*

[8] Android Developer. Monkeyrunner. http://developer.android.com/tools/help/monkeyrunner_concepts.html

[9] Atif M. Memon, Mary Lou Soffa. Regression Testing of GUIs. *ESEC/FSE-11 Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering. Pages 118-127*

[10] Wei Yang, Mukul R. Prasad, and Tao Xie. A Grey-box Approach for Automated GUI-Model Generation of Mobile Applications. *FASE'13 Proceedings of the 16th international conference on Fundamental Approaches to Software Engineering. Pages 250-265*

[11] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Trans. Softw. Eng., 22(8):529–551, Aug. 1996.*

[12] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Meth., 6(2):173–210, Apr. 1997*

[13] Tanzirul Azim, Iulian Neamtiu. Targeted and Depth-first Exploration for Systematic Testing of Android Apps. *OOPSLA '13 Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications. Pages 641-660*

[14] Jonathan Crussell, Clint Gibler, and Hao Chen. Attack of the Clones: Detecting Cloned Applications on Android Markets. *17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings*