1J-08

Generating Test Cases for GUI-based Android Applications by Modeling and Change Analysis

Jose Lorenzo San Miguel　　Shingo Takada
Keio University

1. INTRODUCTION

The popularity of Android application (apps) has grown significantly. Having a short time-to-market release presents a major challenge for app developers. Before developers release their apps, testing is important to ensure the quality of their products.

Android application testing is still a major challenge to app developers. There are a number of existing automated GUI testing methods and tools for Android that have already been studied, implemented, and evaluated [3, 4, 6, 8]. However, according to a study by Joorabchi, et al. [5], 64% of the survey respondents preferred to use manual testing in practice. On the other hand, existing test automation techniques are meant to be used with a single version of the app. To the best of our knowledge, existing testing methods lacked consideration in adapting to changes made to the apps.

Inspired by these findings, we propose an approach to test case generation for Android GUI testing that resembles manual testing by using state models and supports the evolution of apps through change analysis. This paper gives an overview of our on-going work which is the continuation of our previous proposal [7].

The rest of the paper is organized as follows: Section 2 contains related works, Section 3 presents our methodology, and Section 4 states our conclusions.

2. RELATED WORKS

There are a number of works related to Android GUI testing. Gomez, et al. [4] created a tool called *RERAN*, or record-and-replay, which allows captured low-level event streams to be replayed with accurate timing. Choi, et al. [3] proposed using machine learning to create an abstraction of the GUI and to achieve code coverage quickly. The key point of their approach is to reduce the number of app restarts and to guide the test execution without the need of the precise model of the app. Linares-Vásquez, et al. [6] mined app usages to create executable test scenarios. They created a tool coined as *MonkeyLab*, a framework based on Record-Mine-Generate-Validate. Their primary focus was to create executable scenarios that would resemble manual testing. Yang, et al [8] implemented a static analysis tool called *GATOR*. Their work formulated algorithms to construct Window Transition Graph (WTG) which is directly applicable for GUI model construction for program understanding, testing, and dynamic exploration.

“Generating Test Cases for GUI-based Android Applications by Modeling and Change Analysis”
† Jose Lorenzo San Miguel・Keio University
‡ Shingo Takada・Keio University

3. METHODOLOGY

Our approach is based on the following five components: *GUI Modeling, Usage Modeling, Event Sequence Analysis, Test Case Generation,* and *Change Analysis.* System architecture is shown in Figure 1.

3.1 GUI Modeling

This step models an approximate behavior of the app using a state machine. Each activity (e.g., login and home page) is represented as a state while events (e.g., button click) are represented as state transitions. The *GUI Model* is created from the output of a third-party tool called *GATOR* [1]. Our approach gathers the *feasible paths* of the AUT (Application Under Test) generated by *GATOR*. The resulting *feasible paths* are used to create the *GUI Model*.

3.2 Usage Modeling

The goal of this step is for our method to resemble manual testing by collecting normal end-user usage. To gather the app usages, an *Agent* runs in the background and monitors the app while it is being used. The usage logs generated by the *Agent* are collected to output the *Usage Models*. These are state machine representations of user-generated actions. The *Usage Models* are combined with the *GUI Model* to obtain a single state machine called *Behavioral Model*.

3.3 Event Sequence Analysis

We define an *Event Sequence* as the code execution path relative to the state transitions in the *Behavioral Model.* The required inputs for this method are the *Behavioral Model* and the AUT’s call graph called *Structural Model.* The call graph is created by using an existing framework called *Soot* [7]. Figure 2 illustrates an example of *Event Sequence Analysis* processing.

To create an *Event Sequence*, consider the path *{A}–E1→{A}–E2→{B}* in the *Behavioral Model.* Events *E1* and *E2* correspond to an event in the Android context. Corresponding event handlers for these events (i.e., *E1H* and *E2H* in *Structural Model*) are used as entry

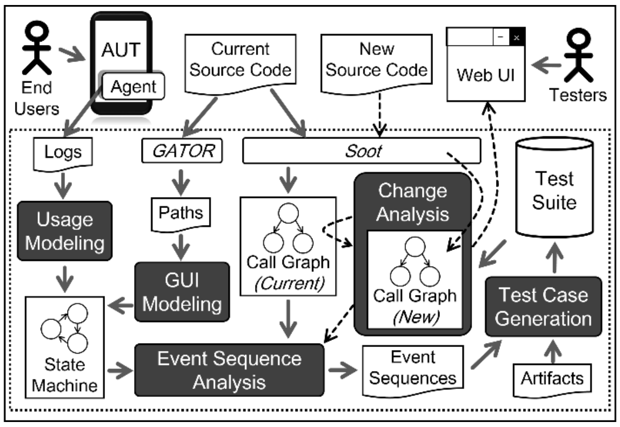

Figure 1: System Architecture

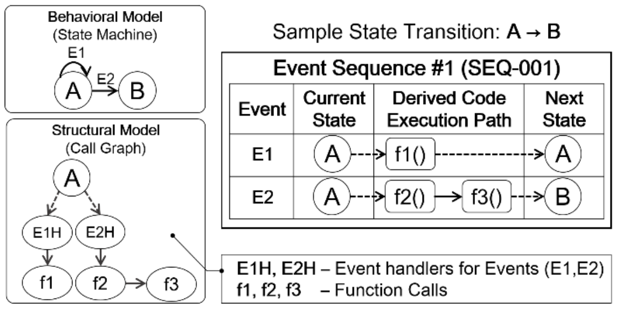
Behavioral Model (State Machine)
E1
E2
A
B
Structural Model (Call Graph)
A
E1H
E2H
f1
f2
f3
Sample State Transition: A → B
Event Sequence #1 (SEQ-001)
Event
Current State
Derived Code Execution Path
Next State
E1
A
f1()
A
E2
A
f2()
f3()
B
E1H, E2H – Event handlers for Events (E1,E2)
f1, f2, f3 – Function Calls

Figure 2: Example of Event Sequence Analysis

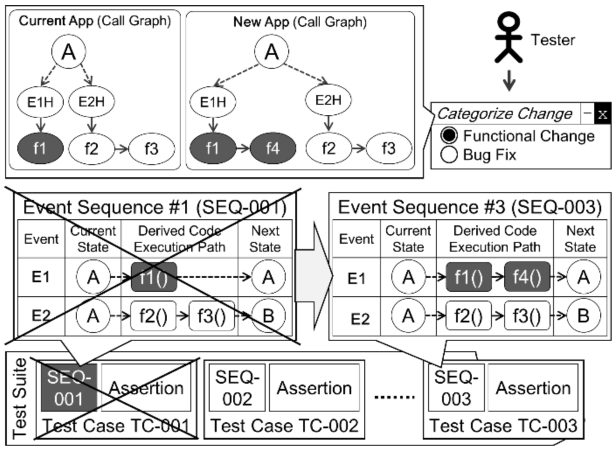

Figure 3: Example of Change Analysis

points for the code execution.

Using the *Structural Model*, this analysis derives the code execution path, which starts from event handlers *E1H* and *E2H*. In Figure 2, *E1H* executes function *f1()* while *E2H* executes *f2()* and then *f3()*.

3.4 Test Case Generation

Our work defines a test case as the combination of an *Event Sequence* and a desired output called *Assertion*. Test values are retrieved from an artifacts database which contains a base of “what to test” items (e.g., text field values).

3.5 Change Analysis

Our approach uses *Change Analysis* to compare the differences of the *Structural Models* of the current and new apps. Our work includes static analysis techniques to determine the call graph nodes that were affected by the changes. This change information is made available to testers via a *Web UI* for them to categorize the changes. Changes are categorized into *Functional Change* (may require update of test cases) or *Bug Fix* (test cases must not be updated but needs to be retested). In our approach, *Change Analysis* is done once *Test Case Generation* for the current app has been completed. Figure 3 shows an example of this method.

In the given call graphs, change was made in function *f1()* by adding a function call to *f4()* and was categorized as *Functional Change*. From the list of existing event sequences, we analyzed that *SEQ-001* was affected by the change since it calls the updated function *f1()*. Hence, *SEQ-001* is invalidated.

From the *Test Suite*, our method determines the test cases that use the affected event sequence (i.e., *SEQ-001*). In the example, test case *TC-001* uses *SEQ-001*. Therefore, we consider *TC-001* as an invalid test case.

To cope with the changes, *Event Sequence Analysis* and *Test Case Generation* is performed again. In the given figure, *SEQ-003* is the new event sequence while *TC-003* is generated using *SEQ-003*.

4. CONCLUSION

We presented our approach to test case generation, which uses includes *GUI Modeling, Usage Modeling, Event Sequence Analysis*, and *Change Analysis*.

5. REFERENCES

[1] GATOR. http://web.cse.ohio-state.edu/presto/software/gator/. (Last accessed: 11-Nov-2015).
[2] Soot. http://sable.github.io/soot/. (Last accessed: 25-Dec-2015).
[3] W. Choi, et al. Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning. In *OOPSLA'13*, pages 623–640, 2013.
[4] L. Gomez, et al. RERAN: Timing- and Touch-Sensitive Record and Replay for Android. In *Proc. of ICSE'13*, pages 72–81, 2013.
[5] M. E. Joorabchi, et al. Real Challenges in Mobile App Development. In *Proc. of ESEM'13*, pages 15–24, 2013.
[6] M. Linares-Vásquez, et al. Mining Android App Usages For Generating Actionable GUI-based Execution Scenarios. In *Proc. of MSR'15*, pages 111–122, 2015.
[7] J. L. San Miguel, et al. Generating Test Cases for Android Applications through GUI Modeling, Usage Modeling, and Change Analysis. In *C3S2E15*, pages 146–147, 2015.
[8] S. Yang, et al. Static Window Transition Graphs for Android. In *Proc. of ASE'15*, 2015.