Towards Realistic Mobile Test Generation

Ke Mao*
CREST Centre, Department of Computer Science
University College London, Malet Place, London, WC1E 6BT, UK
k.mao@cs.ucl.ac.uk

ABSTRACT

Generating realistic tests is a major challenge of testing mobile applications. Most existing mobile test generation techniques do not optimise realism of tests, which may generate test cases with undesired properties, such as over long sequences, too frequent events, unnatural sequence patterns. These unrealistic tests introduce difficulties to developers when debugging. This work proposes an approach to generate realistic tests with a natural/non-intrusive execution framework. The approach aims to automate the real-world manual mobile testing process performed by test experts and end users, while revealing faults with generated realistic tests.

CCS Concepts

•**Software and its engineering** → **Software testing and debugging;** *Search-based software engineering;*

Keywords

mobile testing; realistic; search-based testing;

1. INTRODUCTION

Accompanying the wide-spread growth in mobile device ownership and the number of mobile apps [2,3], mobile testing activities are highly demanded. However several new features pose emerging challenges to mobile app testing: Mobile devices enable rich user interaction inputs such as gestures via touch screens and various signals via sensors (GPS, accelerometer, barometer, NFC, etc.). They serve a wide range of users in heterogeneous and dynamic contexts such as geographical locations and networking infrastructures. Complex interactions with various sensors under a wide range of testing contexts are required. A recent survey work on mobile app development indicates that current mobile app testing in practice heavily rely on manual testing [20]. Although a few frameworks from the industry (such as Appium [1], Robotium [4], UIAutomator [5]) can support the automatic execution of test cases. They rely on human to write the testing scripts. Moreover, it is expensive to maintain these scripts as the apps under development are subject to frequent changes.

Many automated testing techniques have been proposed in Android testing research [8,9,14,16,24,25,27,28,29], however all of these techniques use intrusive approaches to execute the generated test cases, requiring developer permissions to perform the testing which is not always available. Many of them also need to modify the app code or even the mobile systems. This may partially explain why the open-sourced Android platform were used in these research, rather than iOS or other mobile platforms.

This work focus on realistic test generation with non-intrusive test execution. It aims to address the following research questions:

RQ1 (Realism differential): What are the differences on realism, between the test cases that are automatically generated (by the state-of-the-art approaches) and those are generated by real-world test experts or users?

RQ2 (Realism modelling): Can we model realistic tests by proposing a set of realism metrics (e.g., test event type, event frequency, event parameter, sequence length, sequence pattern, replicablity) to capture their characteristics?

RQ3 (Realistic test generation): Can we automatically generate realistic test cases by searching a refined solution space, based on the realistic test case model?

RQ4 (Impact evaluation): What are the differences on the fault detection capability, between automatically generated realistic and unrealistic test cases? How do developers values the faults identified by corresponding types of test cases?

The hypothesis for this work is that automatically generated test cases generally differ from those used by real-world testing (e.g., manual exploratory testing, end-user testing), when assessed from a set of '*realism*' metrics. These 'unrealistic test cases' can be less valuable to developers even if they can reveal faults. Also, previous intrusive execution of these test cases are unnatural and cannot satisfy certain testing scenarios. By using a 'realism' guided search, realistic test cases can be generated and executed in a natural way to simulate the testing that happened in the real-world.

Expected contributions. This work are expected to contribute on the following aspects:

1) A set of novel metrics to model the realism of test cases;

*Ke Mao is supervised by Prof. Mark Harman and Prof. Licia Capra, and he also works with Dr. Yue Jia.

DOI: 10.475/123_4

2) The empirical evidence on the role of realism in test generation;

3) The empirical evidence on the necessary application scenarios of non-intrusive testing, which is neglected in previous research;

4) A search-based approach to realistic test generation. The approach uses genetic improvement to extend existing test cases and further search for new test cases via a refined search space with realism constraints. The execution of the generated test cases are performed in a non-intrusive way to further simulate real-world testing performed by test experts and users.

2. RELATED WORK AND MOTIVATION

There are several existing studies and tools for mobile testing. A taxonomy of mobile testing is shown in Figure 1.

2.1 Mobile Test Generation Techniques

Mobile test generation aims to automatically generate and execute test inputs, to help developers detect potential failures and fix them before the release of the app. These inputs are usually events as mobile apps are event-driven. The events can be either system events such as screen-shooting, volume-adjusting, or UI events such as clicks and gestures. Existing mobile test generation techniques and tools basically fall into four categories. Most of these research work are based on the Android platform due to its open-source nature and dominating market share.

Record-replay approaches: Record-replay approaches generate realistic test cases, however they require human effort. *SPAG* [21] is a record-replay technique for testing mobile apps. It is backed with event batch and smart wait features for accurately and reliably reproducing the recorded tests. *SPAG-C* [22] further extends *SPAG* by using computer vision techniques to perform test oracle comparison.

Random approaches: Random strategies are widely used in practice for fuzz testing. Android *Monkey* [15] is a built-in tool of the Android platform, which is developed by Google. It randomly seeds various types of Android events into the system. *Dynodroid* [24] also randomly explore the app, but with two designed strategy call BiasedRandom and Frequency. The BiasedRandom strategy adjusts the weights of different events by considering the contexts that the events are associated with. The Frequency strategy enables the generation of events that are least frequently selected. *Dynodroid* supports the generation of both UI and system events. Its implementation is publicly available.

Model-based approaches: *GUIRipper* [9] crawls the GUI of an app. It constructs a model of the user interface dynamically, by dumping the UI element on each execution states. The traversal strategy of the crawler follows the depth-first-search (DFS) algorithm. The limitation of this tool is that it cannot generate system events. This tool is public but not open-sourced. It is known as *MobiGUITAR* [8] later. *PUMA* [16] is a framework for implementing various strategies to explore the app based on a finite state machine (FSM) representation. It contains the Monkey exploration strategy and can be easily extended by directing the transition of the FSM model. *SwiftHand* [14] also dynamically builds a FSM model of the GUI. Its contribution focuses on minimizing the restart times when exploring the app, while maximizing the test coverage.

Systematic approaches: *Sapienz* [27] is a recently proposed technique for multi-objective test generation, which minimises test sequence length while maximising coverage and fault revelation. It uses a combination of high-level 'motif genes' and low-level 'atomic genes' to effectively explore the app. *EvoDroid* [25] is the first search-based testing framework Android apps. It extracts both interface models vis static resource analysis and call graph models via code analysis. It regards an event sequence as an individual to perform genetic make-up and generates test inputs based on an evolutionary algorithm. *ORBIT* [29] is designed to generate only relevant UI events. It achieves this by using both GUI ripping and static code analysis techniques. This tool is not public available. *ACTEve* [10] is based on symbolic execution. It mitigates the path explosion problem by identifying subsumption conditions among test sequences. A^3E [11] implements two app exploration strategies: the depth-first search strategy and the targeted strategy to desired activities of the app. The targeted strategy is based on taint analysis, to construct the app's static activity transition graph. Only the first strategy is available in its publicly available implementation.

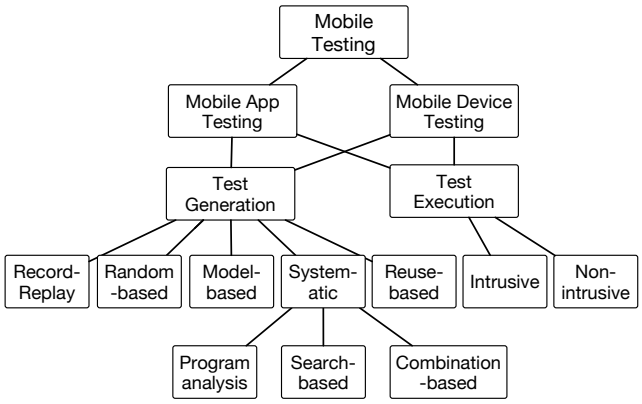

Figure 1: A taxonomy of mobile testing.

Reuse-based approaches: Previously manually written or automated generated test cases are reused for deriving new tests which may be executed in various background conditions. Thor [6] itself does not generate new test cases. Instead, it reuses existing test cases and executes them in adverse conditions, such as sensor status changes or mobile operation system inference.

2.2 Realistic Test Generation

Generally, there is a lack of research on realistic test generation, especially for mobile testing. For Android testing, *MonkeyLab* [23] generates test cases based on the app usage data. There are a limited number of previous studies aiming at realistic test generation, however they are designed for testing web applications or Java programmes [7, 12, 13].

Furthermore, there is a neglect on realistic test execution of the automatically generated test cases. When executing these generated test cases, existing techniques usually adopt intrusive approaches. Intrusive approaches require modification behaviours on either the app under test or the mobile operation system or both of them. The actions generated by this kind of testing are though simulated signals rather than those triggered via real sensors (e.g., touch-screen, gravity-sensor) on the mobile device. The advantages of the intru-

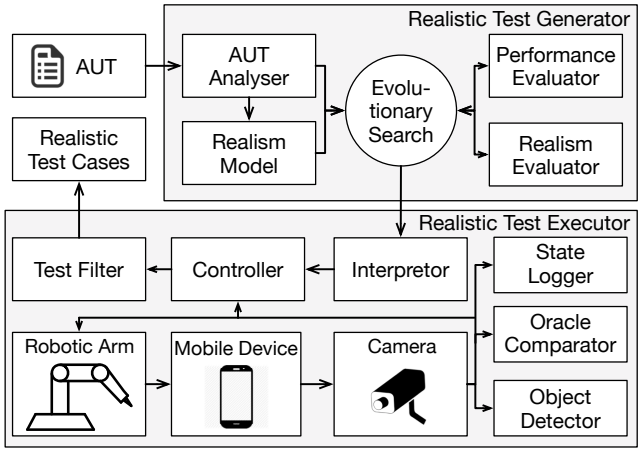

Figure 2: A framework for realistic test generation.

sive approaches are obvious regarding the simulated operation signals: First, it is easier to simulate the operation signal rather than to actually trigger them via hardware. Second, simulation leads to almost no cost on instrumentation and execution hardware. Third, simulated signals are reliable and can be executed accurately. There are also several limitations on intrusive testing: First, the necessary precondition on modifying the mobile system (or requiring developer/root permission) cannot be always satisfied in real-world testing scenarios (e.g., testing a safety-critical system). Second, it is not suitable when testing the mobile device itself or the mobile operation system itself. Third, since the operation signal is simulated, the testing on the coordination between the hardware and the app cannot be covered. Fourth, one simulated signal (e.g., app switch) may require several operation steps in the real-world. It is possible that a serial of frequent simulated signals could never happen in real-world testing.

3. APPROACH AND EVALUATION

In this section, we present the proposed approach in addressing our research questions. The approach is consisted of two stages: realism modelling regarding RQ1 and RQ2 and realistic test generation regarding RQ3 and RQ4.

3.1 Realism Modelling

To answer RQ1, we propose to conduct an empirical study on analysing the realism differences between the automatically generated test cases and those manually generated by test experts and end-users. We first collect these two types of test cases: We collect automatically generated test cases from the state-of-the-art techniques on mobile testing. We collect potential realistic test cases from three sources: by mining usage traces from real-world end users; by mining manual test events generated by real-world test experts; and by reusing existing test cases written by testers. To validate the realism on these two types of test cases, we ask developers to manually label the realism of the test cases. Inter-rater agreement is used to ensure that the rating is reliable. Then we compare the labelled data to see if there is any difference on the realism rating of these two types of test cases.

In addressing RQ2, we first calculate a comprehensive list of properties of test cases, and compare the differences between realistic and unrealistic test cases. Further statistical analysis on the dataset is conducted. Hence significant factors that corresponds to the realism of test cases can be highlighted. We further propose a model to capture the realism underlying the real-world test cases. For example, string realism as part of the model, can be captured by a NLP model to generate readable strings [7].

3.2 Realistic Test Generation

In addressing RQ3, we propose a framework for realistic test generation, as shown in Figure 2. The framework contains two high-level components: the 'realistic test generator' for generating realistic test case candidates, and the 'realistic test executor' for further filtering the test cases, depending on whether they are executable in a real-world setting.

Realistic test generator. The realistic test generator starts by analysing the application under test (AUT). The extracted information of the AUT is used to adjust the realism model. The realism model together with the AUT are passed into the evolutionary search component for generating and evolving test cases. The source of 'realism' for the individuals being evolved comes from two ways: first, by reusing and extending realistic test cases (e.g., Robotium or Appium test scripts) manually written by the app testers; second, by searching a solution space constrained by the realism model. For the first case, we use genetic improvement to extend these existing test cases. For the second case, the realism model is used together with a mobile event generator to guide the search toward generating test sequences with realistic properties. The fitness of the generated potential realistic test cases are evaluated based on their performance (such as code coverage) and realism rating (which may be a suitable task to be crowdsourced [26]).

Realistic test executor. The generated realistic test case candidates are further validated by actually executing them in a real-world/physical way, which is as natural as those performed by end-users or manual testers. The realistic test executor first interprets the coded test scripts into machine executable commands. The controller receives the commands and executes them on a robotic arm. The arm interacts with the mobile device non-intrusively. This process needs *Inverse Kinematics* and *Calibration* in order to make the arm act accurately. A camera is used to monitor the mobile device states. Collected image data is further processed via computer vision techniques for *Object Detection* and *Oracle Comparison*. The overall process data logged in the execution process is finally sent to a test filter for judging whether the candidate test case should be filtered out.

The evaluation of the proposed approach will be conducted on three types of subjects: real-world open source apps, closed source apps and cross platform apps. We will generate realistic test cases by using our approach and generate unrealistic test cases by using the state-of-the-art approaches for mobile testing. A set of performance metrics will be measured to assess the test cases. The assessment will not only be based on the performance metrics but also based on the their values to developers: we will conduct an empirical study by sending the revealed faults together with the corresponding test cases to developers, to see which techniques can generate more developer favoured test cases. The experimental results and the empirical study will lead to the answer for RQ4.

4. RESEARCH STATUS

This work is planed as a part of my PhD thesis. Before this work, we have performed a brief survey on mobile app testing and a comprehensive survey on relevant techniques that may help to generated realistic test cases. We have investigated the use of search-based techniques [17,18,19] for generating test cases for mobile apps. Promising results have been achieved [27]. We have also implemented a prototype of the 'realistic test executor' described in Section 3.2.

5. CONCLUSION AND FUTURE WORK

In this work, we proposed to generate realistic test cases for mobile testing. The first stage of this work aims to model the realism underlying real-world test cases; the second stage focuses on generating realistic test cases based on the built model and further execute them in a non-intrusive way. Future work will be directed towards implementing the proposed approach and conducting studies to answer the proposed research questions.

6. ACKNOWLEDGMENTS

Ke Mao is funded by the UCL GRS and the UCL ORS scholarships. This work is also supported by the Dynamic Adaptive Automated Software Engineering (DAASE) programme grant (EP/J017515).

7. REFERENCES

[1] Appium: Automation for iOS and Android apps. http://appium.io.

[2] Global PC sales fall to eight-year low. http://www.statista.com/chart/4231/global-pc-shipments.

[3] Global smartphone shipments forecast from 2010 to 2019. http://www.statista.com/statistics/263441/global-smartphone-shipments-forecast.

[4] Robotium: User scenario testing for Android. https://github.com/RobotiumTech/robotium.

[5] UIAutomator. https://developer.android.com/tools/testing-support-library/index.html.

[6] C. Q. Adamsen, G. Mezzetti, and A. Møller. Systematic execution of Android test suites in adverse conditions. In *Proc. of ISSTA'15*, pages 83–93, 2015.

[7] S. Afshan, P. McMinn, and M. Stevenson. Evolving readable string test inputs using a natural language model to reduce human oracle cost. In *Proc. of ICST'13*, pages 352–361, March 2013.

[8] D. Amalfitano, A. Fasolino, P. Tramontana, B. Ta, and A. Memon. MobiGUITAR: Automated model-based testing of mobile apps. *IEEE Software*, 32(5):53–59, 2015.

[9] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using GUI ripping for automated testing of Android applications. In *Proc. of ASE'12*, pages 258–261, 2012.

[10] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *Proc. of ESEC/FSE'12*, pages 59:1–59:11, 2012.

[11] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of Android apps. In *Proc. of OOPSLA'13*, pages 641–660, 2013.

[12] M. Bozkurt and M. Harman. Automatically generating realistic test input from web services. In *Proc. of SOSE'11*, pages 13–24, 2011.

[13] M. Bozkurt and M. Harman. Optimised realistic test input generation using web services. In *Proc. of SSBSE'12*, pages 105–120, Riva del Garda, Italy, 2012.

[14] W. Choi, G. Necula, and K. Sen. Guided GUI testing of android apps with minimal restart and approximate learning. In *Proc. of OOPSLA'13*, pages 623–640, 2013.

[15] Google. Android Monkey. http://developer.android.com/tools/help/monkey.html.

[16] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan. PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps. In *Proc. of MobiSys'14*, pages 204–217, 2014.

[17] M. Harman. The current state and future of search based software engineering. In *Proc. of FOSE'07*, pages 342–357, 2007.

[18] M. Harman and B. F. Jones. Search-based software engineering. *Information and software Technology*, 43(14):833–839, 2001.

[19] M. Harman and P. McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering*, 36(2):226–247, 2010.

[20] M. E. Joorabchi, A. Mesbah, and P. Kruchten. Real challenges in mobile app development. In *Proc. of ESEM'13*, pages 15–24, 2013.

[21] Y.-D. Lin, E.-H. Chu, S.-C. Yu, and Y.-C. Lai. Improving the accuracy of automated GUI testing for embedded systems. *IEEE Software*, 31(1):39–45, 2014.

[22] Y.-D. Lin, J. Rojas, E.-H. Chu, and Y.-C. Lai. On the accuracy, efficiency, and reusability of automated test oracles for Android devices. *IEEE Transactions on Software Engineering*, 40(10):957–970, October 2014.

[23] M. Linares-Vásquez, M. White, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk. Mining Android app usages for generating actionable GUI-based execution scenarios. In *Proc. of MSR'15*, 2015.

[24] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for Android apps. In *Proc. of ESEC/FSE'13*, pages 224–234, 2013.

[25] R. Mahmood, N. Mirzaei, and S. Malek. EvoDroid: Segmented evolutionary testing of Android apps. In *Proc. of ESEC/FSE'14*, pages 599–609, 2014.

[26] K. Mao, L. Capra, M. Harman, and Y. Jia. A survey of the use of crowdsourcing in software engineering. Technical Report RN/15/01, Department of Computer Science, University College London, 2016.

[27] K. Mao, M. Harman, and Y. Jia. Sapienz: Multi-objective automated testing for Android applications. In *ISSTA'16*, 2016. to appear.

[28] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood. Testing Android apps through symbolic execution. *SIGSOFT Software Engineering Notes*, 37(6):1–5, 2012.

[29] W. Yang, M. R. Prasad, and T. Xie. A grey-box approach for automated GUI-model generation of mobile applications. In *Proc. of FASE'13*, pages 250–265, 2013.