2015 IEEE/ACM 37th IEEE International Conference on Software Engineering

Enabling Testing of Android Apps

Mario Linares-Vásquez
The College of William and Mary, Williamsburg, VA, USA
mlinarev@cs.wm.edu

***Abstract*—Existing approaches for automated testing of Android apps are designed to achieve different goals and exhibit some pros and cons that should be carefully considered by developers and testers. For instance, random testing (RT) provides a high ratio of infeasible inputs or events, and test cases generated with RT and systematic exploration-based testing (SEBT) are not representative of natural (*i.e.*, real) application usage scenarios. In addition, collecting test scripts for automated testing is expensive. We address limitations of existing tools for GUI-based testing of Android apps in a novel hybrid approach called T+. Our approach is based on a novel framework, which is aimed at generating actionable test cases for different testing goals. The framework also enables GUI-based testing without expensive test scripts collection for the stakeholders.**

I. Motivation

Each approach for GUI-based testing of mobile apps can be applied to achieve a specific testing goal. However, manual testing is still preferred and relied upon over automated testing [9], despite seemingly lower coverage statistics that are achieved with manual testing. This suggests that an automatic testing framework must simulate the convenience and naturalness of manual testing, while requiring little developer effort and avoiding the complexity generally associated with automated testing tools. In general, creating such a testing framework should at least (i) generate actionable test cases (*i.e.*, streams of feasible events that can be reproduced automatically on different devices, (ii) have a context awareness of an Application Under Test's (AUT) execution history and current possible actions, and (iii) should not require expensive test scripts collection. However, current approaches partially cover these requirements.

II. Related Work

State-of-the-art approaches for GUI-based testing exhibit some pros and cons that should be carefully considered:

1) Fuzz/Random testing: This type of testing does not require prior-knowledge about the AUT and is easy to set up, however, it can flood AUT's GUI with infeasible events. The most used tool for this approach in Android apps is the Android `UI monkey` [7].

2) Systematic Exploration-Based Testing (SEBT): The exploration of the AUT is driven automatically by a GUI model that describes the screens, GUI components, and feasible actions on each component. Such a model can be generated (manually or automatically) before the exploration, or extracted iteratively after a new event is executed. Because the main goal of SEBT is to execute systematically all the GUI-components, this is considered to be a good approach for achieving high coverage. However, the exploration is not representative of natural (*i.e.*, real) application usage scenarios. Examples of strategies for SEBT using a heuristic (*a.k.a.*, systematic exploration, ripping, crawling) are the work by Takala *et al.* [12], and the tools *VanarSena* [11], *AndroidRipper* [2], *ORBIT* [14], A^{3}E [3], and *Dynodroid* [10].

3) Record and Replay (R&R): Although, this is the closest approach to manual testing, R&R requires traces (or testing scripts) to be re-recorded if the GUI changes, and the effectiveness of the script depends on the ability of the approach to represent complex gestures. Also a trace/script is required for each device, because traces/scripts are often coupled to locations in the screen. *MonkeyRecorder* [1] and RERAN [5] are examples of record-and-replay strategies for Android apps.

4) Event-sequence Generation: Relies on generating streams of tokens (*i.e.*, events) that form a test case. This approach requires a model that can be defined manually or derived automatically. One of the major problems of the event-sequence generation is that some events in the sequence are infeasible in the context of the test case. Representative tools are *SwiftHand* [4], which uses statistical models extracted from execution traces collected a-priori, and *Collider* [8], which combines GUI models and concolic execution to generate tests cases that end at a target location, and the work by Tonella *et al* [13] on generating test cases with language models.

III. Approach

In this section we present our hybrid approach for automated GUI-based testing of Android apps, where we provide solutions to the mentioned problems (Section I). Our approach is described by the `Record`$\rightarrow$`Analyze`$\rightarrow$`Generate`$\rightarrow$`Validate` framework: (i) developers/testers use the AUT naturally; and low-level event logs representing scenarios executed by the developers/testers are `Record`ed in the device; (ii) the logs are `Analyze`d to obtain event sequences decoupled from exact screen coordinates; (iii) the source code of the AUT and the event sequences are `Analyze`d to build a vocabulary of feasible events; (iv) models representing the GUI and behavior are derived using the vocabulary of feasible events; (v) the models are used to `Generate` candidate test cases; and (vi) the test cases are `Validate`d on the target device where infeasible events are removed for generating *actionable test cases (i.e., feasible and fully reproducible)*. We coined this framework as **T+**. The architecture is depicted in Figure 1, and the four phases are described with more details in the following.

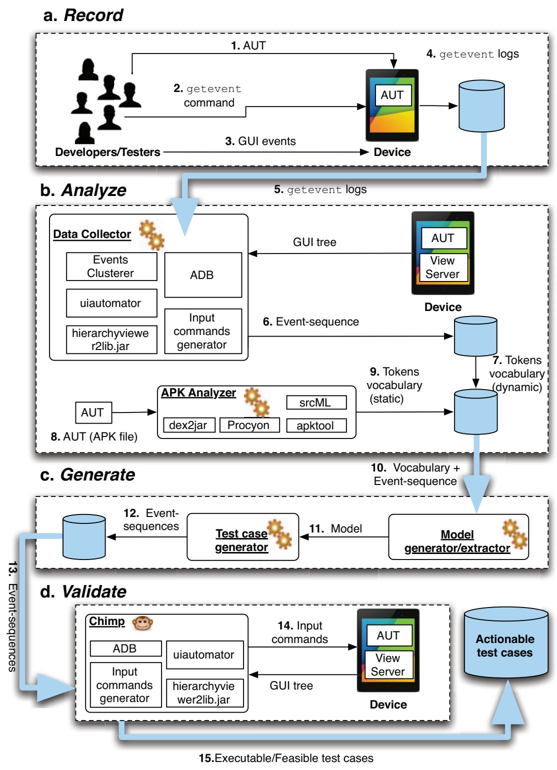

Fig. 1. Architecture of the **T+** framework for GUI testing

`Record`. Collecting event logs from developers/testers should be no different from manual testing (in terms of experience and overhead). Therefore, **T+** relies on the `getevent` command for collecting event sequences, similar to *RERAN* [5]. The `getevent` command enables logs to include low-level events that represent click-based actions, simple (*e.g.*, swipe) and complex gestures; those logs are naturally collected during an AUT's execution. After **T+** is enabled, developers/testers exercise/test the AUT as in manual testing. After having executed the app, the logs are generated and copied to the logs repository (Figure 1-a). Since our log collection approach poses no overhead on developers/testers, this setup permits collecting logs on a large scale. In fact, this log collection approach can be easily crowd-sourced, where we can also collect logs from daily app usages by ordinary users.

`Analyze`. This phase is aimed at extracting the vocabulary of events required to build the test-case-generation model. The vocabulary of events is extracted from the source code of the AUT and also from automatic reproduction of collected logs (Figure 1-b); by combining static and dynamic analyses we increase the universe of components and possible actions that can be analyzed for the test case generation. Because, the low-level events are coupled to specific locations on the screen, we designed a method for translating the events descriptions to high-level representations. Therefore, low-level events were translated to tuples e_i :=<*Activity*$_i$, *Window*$_i$, *GUI-Component*$_i$, *Action*$_i$, *Component-Class*$_i$>. For the translation, we replayed the events in a step-by-step mode, and for each step we identified the corresponding GUI element from the GUI-tree.

`Generate`. To provide developers/testers with a multi-goal testing framework, our framework should derive test cases from different models. For example, if the goal is to generate unnatural test cases (*e.g.*, for fuzz testing) the underlying model could be a probabilistic distribution over the universe of possible events, which selects unseen events in the traces collected; if the goal is maximizing coverage, systematic exploration could be used. **T+** is independent of the underlying model for generating test cases; therefore the philosophy behind the `Generate` phase (Figure 1-c) is that instances of different models can be plugged into **T+**. Meanwhile **T+** provides the models with the events vocabulary, the expected output of each model are sequences of tuples e_i.

`Validate`. The streams of events generated by the *Test case generator* are expressed at GUI level. This enable the test cases to be translated to any type of input commands and to execute the test case in a step-by-step mode. Therefore, the test cases can be translated to input commands that can be executed remotely on a device, test units written using the Android UI Automator API, or other script-based commands for executing GUI events on Android apps. Because the models used to generate the test cases can generate infeasible events in the sequence, **T+** validates each step on a real device, and filters from the test cases the infeasible events. The filtering and test case translation to a specific language, is done by a software component in **T+** coined as *Chimp* (Figure 1-d). Our *Chimp*, translates event-sequences to ADB input commands and relies on the Android UI Automator to recognize whether an event in the test case is feasible.

IV. Results and Contributions

We have used the proposed framework with five Android apps (Diabetes +1.0.,4 Keepscore 1.5.1, Tasks 1.0.12, and Word Web Dict. 2.1) for three testing scenarios: (i) random testing, (ii) systematic exploration using a depth-first search (similarly to [2], [3], [14], [11]), and (ii) natural/unnatural test cases generation by using language models (similarly to [13]). In, the three scenarios we have been able to generate actionable test cases successfully with **T+**. However, during our preliminary experiments we have found some challenges that impact the automatic generation of test cases for Android apps: (i) encapsulated components (*e.g.*, `KeyboardView, AutoCompleteTextView, CalendarView, DatePicker`) can not be analyzed during systematic exploration, because the sub-components (*e.g.*, each key of the keyboard) are not available for ripping at execution time; (ii) testing approaches should be able to deal with clean (*i.e.*, no previous data or cache in the app) and dirty launches (*i.e.*, the app keeps data from previous executions); (iii) the arrival time between events, during real execution of mobile apps, should also be modeled to reduce idle time during testing and to avoid the situation where feasible events become infeasible (one option for dealing with this is by generating test cases written with the Android UI Automation API); and (iv) statistical models require appropriate mechanisms for identifying the required size of training corpus.

References

[1] Android Monkey Recorder. http://code.google.com/p/android-monkeyrunner-enhanced/.

[2] D. Amalfitano, A. Fasolino, P. Tramontana, S. De Carmine, and A. Memon. Using GUI Ripping for Automated Testing of Android Applications. In 258-261, editor, *International Conference on Automated Software Engineering (ASE'12)*, 2012.

[3] T. Azim and I. Neamtiu. Targeted and Depth-first Exploration for Systematic Testing of Android Apps. In *International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'13)*, pages 641–660, 2013.

[4] W. Choi, G. Necula, and K. Sen. Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning. In *International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'13)*, pages 623–640, 2013.

[5] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. RERAN: Timing- and Touch-Sensitive Record and Replay for Android. In *International Conference on Software Engineering (ICSE'13)*, pages 72–81, 2013.

[6] Google. Android Debug Bridge. http://developer.android.com/tools/help/adb.html.

[7] Google. UI/application exerciser monkey. http://developer.android.com/tools/help/monkey.html.

[8] C. S. Jensen, M. R. Prasad, and A. Moller. Automated Testing with Targeted Event Sequence Generation. In *International Symposium on Software Testing and Analysis (ISSTA'13)*, pages 67–77, 2013.

[9] M. E. Joorabchi, A. Mesbah, and P. Kruchten. Real Challenges in Mobile App Development. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'13)*, pages 15–24, 2013.

[10] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An Input Generation System for Android Apps. In *9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'13)*, pages 224–234, 2013.

[11] L. Ravindranath, S. nath, J. Padhye, and H. Balakrishnan. Automatic and Scalable Fault Detection for Mobile Applications. In *12th annual international conference on Mobile systems, applications, and services (MobiSys'14)*, pages 190–203, 2014.

[12] T. Takala, M. Katara, and J. Harty. Experiences of System-Level Model-Based GUI Testing of an Android Application. In *Fourth International Conference on Software Testing, Verification and Validation (ICST'11)*, pages 377–386, 2011.

[13] P. Tonella, R. Tiella, and C. Nguyen. Interpolated N-Grams for Model Based Testing. In *International Conference on Software Engineering (ICSE'14)*, 2014.

[14] W. Yang, M. Prasad, and T. Xie. A Grey-box Approach for Automated GUI-Model Generation of Mobile Applications. In *16th International Conference on Fundamental Approaches to Software Engineering (FASE'13)*, pages 250–265, 2013.