ICSE: G: Enabling Testing of Android Apps

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

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 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.

1. PROBLEM AND MOTIVATION

Although several tools are available to support automated execution of Android apps for validation purposes, in practice, testing is still performed mostly manually according to recent survey studies [21, 22, 25]. This preference is supported by the lack of tools that reflect developer/tester needs, continuous pressure for delivery from the market, fragmentation of devices and OS, and lack of resources (e.g., time, testers, devices) for conducting effective testing, among others [21, 22, 25]. Available tools for automated GUI testing of mobile apps, partially solve some of theses issues by (i) proposing mechanisms for recording-and-replaying (R&R) testing scenarios, and (ii) deriving test sequences from models that partially represent the GUI and behavior of apps. However, despite availability of several R&R tools and automation APIs (e.g., Espresso, UIAutomator), a significant amount of work is required to generate replay/testing scripts that are coupled to screen dimensions for a single device (i.e., one script is required for each target device), or have predefined testing sequences. Consequently, these scripts become quickly outdated when significant changes are made to the app's GUI [21, 18].

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

In the case of test sequences derived from models, these models are abstract representations that can be decoupled from device dimensions and event locations and can still remain valid when small changes are done to the app (e.g., button location change). However, current approaches fall short in (i) generating scenarios that are representative of natural (i.e., typical end-user) application usages [33]; (ii) taking into account app's execution history [13, 33]; (iii) generating sequences with previously unseen (i.e., not available in artifacts used to derive the model) but feasible events; (iv) generating sequences that can be executed automatically on different devices. Moreover, utilizing model-based testing techniques for GUI-based testing in industrial contexts may be particularly challenging because creating such models requires specialized expertise and using these models (for manual and automated testing) assumes a complete logical mapping between the model and the actual system that was modeled [4] (which is not always the case).

In this paper we describe a novel approach, **T+**, which aims at solving most of the aforementioned issues of current automated approaches for GUI testing of Android apps. In practice, developers constantly test their apps *manually* by exercising apps on target devices, and real app usages are a rich source of information for extracting a usage (a.k.a., behavioral) model of Android apps without relying on a-priory models that can be outdated. Our main goal with **T+** is to design an automated testing framework that simulates the convenience and naturalness of manual testing while requiring little developer effort and avoiding the complexity generally associated with automated testing tools. Therefore, a key hypothesis under **T+** design *is that all this data that is produced from regular app usages by developers, testers, or even end-users can be effectively recorded and mined to generate representative app usage scenarios (as well as the corner cases) that can be useful for automated validation purposes.* Furthermore, our intuition is that the models mined from execution (event) traces can be augmented with static analysis information to include unseen but feasible events, and other information that can drive the testing process.

T+ is a generic framework that enables combining different models that can be extracted from the code and other sources such as real executions or user reviews, therefore, it can be tailored to achieve different testing goals (e.g., API-driven testing or unnatural test case generation). **T+** analyzes real app usages to derive a usage model (which is not currently considered by available tools), but also statically analyzes the source code to extract a GUI model. A novel feature in **T+** is the generation of test sequences that are

aware of execution history (i.e., previous step) by relying on statistical models. Moreover, **T+** includes a validation step that generates test sequences composed only of feasible steps. Finally, **T+** is able to extract a GUI-level model that is decoupled from components' location and devices' size; this feature enables automatic generation of test scripts for different devices.

2. RELATED WORK

Both industry and academia have provided practitioners with a plethora of tools and approaches for automatic testing of Android apps. The tools can be categorized in five main groups:

- **Fuzz/Random testing (RT):** GUI events are generated randomly and executed on the device. This type of testing does not require prior-knowledge about the application under test (AUT) and is easy to set up, however, it can flood AUT's GUI with infeasible events. The most used tool for RT in Android apps is the Android `UI monkey` [17].
- **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 approaches for SEBT using different exploration heuristics (*e.g.*, Depth-First-Search) are *VanarSena* [32], *AndroidRipper* [5], A^{3}E [8], and *Dynodroid* [27].
- **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], RERAN [14], and VALERA[19] are examples of record-and-replay tools for Android apps.
- **Frameworks for GUI Tests Automation:** Similarly to JUnit that allows for writing test cases using a defined API, in the case of mobile testing, there are several APIs that allow developers/testers to write test cases focused on GUI interaction. Examples of those APIs are the Espresso [15] and UIAutomation [16] projects from Google, and independent projects such as Appium [2] and Roboelectric [3].
- **Event-sequence Generation:** This approach requires a model that can be defined manually or derived automatically; then the model is used to generate streams of tokens (*i.e.*, events) that form a test case. 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. One representative tool is *SwiftHand* [11], which uses statistical models extracted from execution traces collected a-priori. There are also approaches that can be considered as *hybrid*, because they rely on the combination of different models to derive test sequences. For example, *ORBIT* [34] combines static analysis and GUI ripping and *Collider* [20] combines GUI models and concolic execution. Other examples of hybrid approaches are *Evo-Droid* [28], *SIGDroid* [29], *MonkeyLab* [26], and *Crash-scope* [31].

3. APPROACH AND UNIQUENESS

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 1). Our approach is described by the ***Record*** → ***Analyze*** → ***Generate*** → ***Validate*** framework: (i) developers/testers use the Application Under Test (AUT) naturally; and low-level event logs representing scenarios executed by the developers/testers are ***Recorded*** in the device; (ii) the logs are ***Analyzed*** to obtain event sequences decoupled from exact screen coordinates; (iii) the source code of the AUT and the event sequences are ***Analyzed*** 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 ***Validated*** on the target device where infeasible events are removed for generating *actionable test cases (i.e., feasible/fully reproducible)*. We coined this framework as **T+**. The architecture is depicted in Figure 1, and the four phases are described in greater detail in the following.

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* [14]. 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 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 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 are translated to tuples e_i :=<*Activity*$_i$, *Window*$_i$, *GUI-Component*$_i$, *Action*$_i$, *Component-Class*$_i$>. To perform 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, **T+** 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

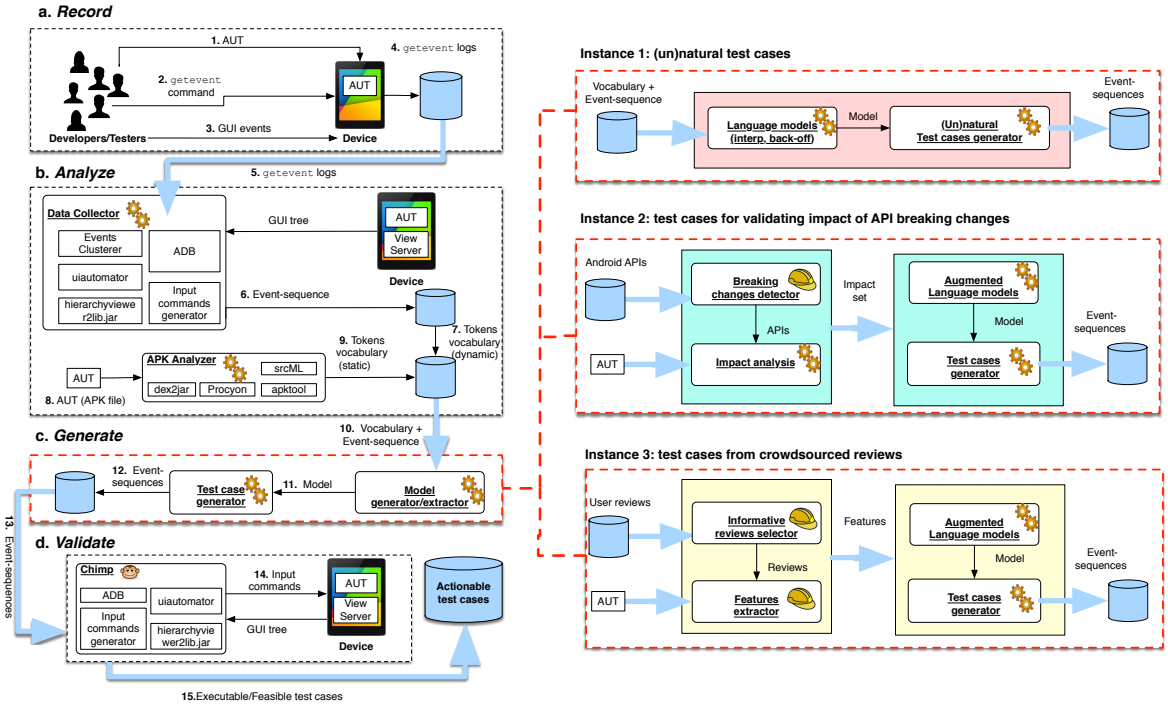

Figure 1: Architecture of the T+ framework for GUI testing

of possible events, which selects unseen events in the traces collected; if the goal is to maximize coverage, systematic exploration could be used. **T+** is independent from the underlying model for generating test cases; thus, 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 enables 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 test cases can potentially generate infeasible events in the sequence, **T+** validates each step on a real device, and filters the infeasible events from the test cases. The filtering and test case translation to a specific language is done by a component 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.

4. RESULTS

The **T+** framework was initially used in a case study with four Android apps (Diabetes +, Keepscore, Tasks, and Word Web Dict.) and unlocked/rooted Nexus 7 Axus tablets each having a 1.5GHz Qualcomm Snapdragon S4 Pro CPU and equipped with Android 4.4.2 (kernel version 3.4.0-gac9222c). In the preliminary experiments, we were able to generate actionable test sequences with three different goals: (i) random testing, (ii) systematic exploration using a depth-first search, and (iii) natural/unnatural test cases generation by using language models. Each testing goal is represented by a specific *Test Case Generator* that is plugged into the infrastructure for the *Generate* phase in **T+**.

For the goal of generating (un)natural test cases, **T+** has been instantiated by combining information extracted from real application usages and the universe of GUI components declared in an app (by using static analysis) [26] (this implementation is called *MonkeyLab*); as for the statistical models that drive the test sequences generation, we used back-off and interpolation n-grams with specific probability distributions we designed to generate (un)natural test sequences [26]. Also, the **T+** components developed for extracting GUI components information from Android apps, have been reused in two approaches for (i) auto-completing bug reports for Android applications [30], and (ii) automatic detection of crashes with a hybrid strategy that combines systematic exploration and static analysis [31].

For the case of *MonkeyLab* [26] we conducted an experiment in which we compared (un)natural test sequences to manual testing, Random Testing, and Systematic Exploration (DFS), in terms of method coverage and GUI events generated by each approach. The experiment was conducted with 5 Android apps (Car Report, Gnu Cash, Mileage, My Expenses, and Tasks) (see [26] for complete details). We analyzed the GUI events generated by each approach, as a way to identify whether *MonkeyLab* is able to generate test sequences that are not exercised by the other approaches and verify the hypothesis that *MonkeyLab* generates (un)natural test sequences. It is worth noting that because of the **T+** framework, *MonkeyLab* combines GUI events extracted from real application usages with the GUI components universe (i.e., GUI model) extracted statically from the source code; therefore, GUI components not exercised manually by humans are present in the test sequences generated by *MonkeyLab*.

Table 1 summarizes the accumulated coverage achieved by each approach, and Figure 2 depicts the number of source

Table 1: Accumulated Statement Coverage of the Analyzed Approaches

App	Users	DFS	Monkey	*MonkeyLab*
Car Report	65%	26%	18%	30%
Gnu Cash	28%	7%	15%	22%
Mileage	61%	14%	38%	26%
My Expenses	33%	10%	38%	26%
Tasks	64%	41%	8%	42%
Average	50.2%	14.0%	23.4%	26.0%

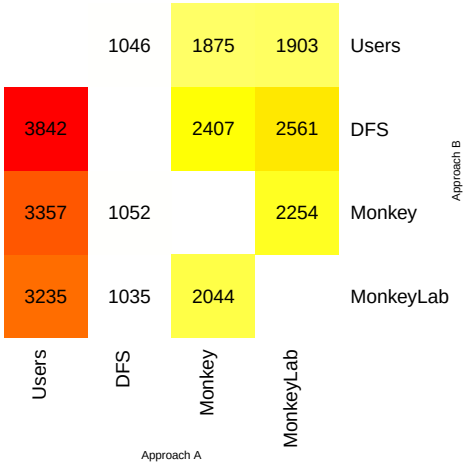

Figure 2: Total number of source code methods in which coverage is higher when comparing coverage achieved by Approach A versus Approach B

code methods in which an approach achieved better coverage compared to another. For computing the coverage we considered (i) 15 minutes of functional testing (per app) performed by five Ph.D. students at the College of William and Mary, (ii) the whole execution of the DFS, and (ii) 100 sequences with 100 events for Monkey and *MonkeyLab*. On average, the test sequences generated by *MonkeyLab* outperformed the approaches for random and systematic testing in terms of coverage. Although the overall coverage achieved by *MonkeyLab* is not as high as compared to manual execution, *MonkeyLab* is able to generate actionable scenarios including not only natural GUI events, but also events that are not considered during manual testing. *MonkeyLab* generates not only natural scenarios but also corner cases that are not generated by any of the competitive approaches. For example, *MonkeyLab* generated 1,611 GUI events not executed by the humans, compared to 1,038 GUI events exercised by the participants that were not generated by *MonkeyLab*. Therefore, the test sequences generated by *MonkeyLab* can help increase the coverage achieved by manual testing without extra effort that is imposed by collecting test-script. Examples of the actionable test sequences can be viewed at the following youtube video list: http://bit.ly/23ED7C6.

5. CONTRIBUTIONS AND FUTURE WORK

The proposed framework and the underlying infrastructure enable several important facets of mobile testing:

A General Infrastructure for Enabling Testing of Android Apps. The *Record* → *Mine* → *Generate* → *Validate* framework is based on a modular design, in which the component in charge of generating the test sequences can be replaced to derive test sequences with different purposes. Figure 1 illustrates the case in which the *Generate* phase can be tailored to different purposes while reusing the components and features from the other phases. *MonkeyLab* [26] is the first instance of this framework which is aimed at generating (un)natural test cases by relying on language models. However, other types of test sequences can be generated using the same infrastructure. For instance, we identified that APIs instability and fault-proneness is a threat to the success of Android apps [23, 24, 9]. When a new API is released, developers should conduct regression testing on the apps to verify that the new API does not inject bugs in the client code (i.e., apps). Therefore, by mining the changes to the APIs and performing static analysis on the source code of the apps, it would be possible to drive generation of test sequences towards executing features that are likely to be impacted by API breaking changes. In addition, informative reviews reporting bug/crashes could be analyzed to generate test cases aiming at testing features/GUI components involved in the bug/crashes reported by users. Both scenarios, might be implemented by augmenting/extending the *Generate* phase in the proposed infrastructure.

Combining Multiple Models For Test Cases Generation. Requirements of a software system are often modeled as a combination of three essential models in Object Oriented (OO) Software Engineering approaches [6, 7]: (i) a *usage model* that describe how users will work with the system (e.g, use cases or user stories); (ii) a *domain model* that describes the business entities, relationships, and data constraints (e.g., class diagram or database model; and (iii) a *GUI model* depicting the user interface. Therefore, those three models are the main sources of information used for designing test cases. In fact, deriving test cases from the usage model is a well promoted practice in OO Software Engineering approaches as a systematic way to validate requirements [10, 12]. For mobile apps, there is a fourth model (*contextual model*) that describes [31] the events/states related to contextual events (e.g., GPS and WiFi) and adversarial conditions.

Few approaches for automated testing of mobile apps have combined two of the models for requirements specifications (i.e., usage, GUI, domain, contextual). For instance, *MonkeyLab* combines usage and GUI models, *SIGDroid* [29] combines GUI and domain models, and *CrashScope* [31][1] combines GUI and contextual models. As of today, there is no complete solution/approach combining the four models (i.e., GUI, usage, domain, and contextual); therefore, future work should be devoted to build solutions for automated testing that combine the aforementioned four models. One potential solution is to reuse the *Record* → *Mine* → *Generate* → *Validate* framework, and to design graphical probabilistic models that are able to consolidate the information from the GUI, usage, domain, and contextual models. The GUI, domain, and contextual models can be extracted statically from the source code in the *Analyze* phase of **T+**.

Automatic Translation of Test Sequences. **T+** derives test sequences from a model that uses tokens at GUI level, which means that the tokens represent high-level actions that are not coupled to specific locations. Therefore, during the *Validate* phase a specific test sequence can be translated into an actionable test case in any test scripting language (including natural language) for any device. The

[1] *CrashScope* is supported by several components developed for the *Analyze* phase of **T+**.

initial implementation of **T+** generates actionable test cases as sequences of ADB input commands, however, we have already added functionality for generating test sequences using the UI Automator and Espresso APIs.

Crowsourced Testing. The `Record` phase in **T+** can be extended to remotely collect event logs from users on a large scale. Future work will be devoted to provide users with an Android app that can be launched on a device for collecting the logs and sending them to a server. A second option is to integrate the Smartphone Test Farm (STF) open source framework (http://openstf.io) into the **T+** infrastructure, so that users can remotely collect traces on devices in a test farm by simply interacting with their browsers.

6. REFERENCES

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

[2] Appium. http://appium.io.

[3] Roboelectric. http://robolectric.org.

[4] P. Aho, M. Suarez, T. Kanstren, and A. Memon. Murphy tools: Utilizing extracted gui models for industrial software testing. In *ICSTW'14*, pages 343–348, 2014.

[5] 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, *ASE'12*, 2012.

[6] S. Ambler. *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process*. John Wiley & Sons, Inc., New York, NY, USA, 2002.

[7] S. W. Ambler. *The Object Primer: Agile Model-Driven Development with UML 2.0*. Cambridge University Press, New York, NY, USA, 2004.

[8] T. Azim and I. Neamtiu. Targeted and Depth-first Exploration for Systematic Testing of Android Apps. In *OOPSLA'13*, pages 641–660, 2013.

[9] G. Bavota, M. Linares-Vásquez, C. E. Bernal-Cárdenas, M. D. Penta, R. Oliveto, and D. Poshyvanyk. The impact of api change- and fault-proneness on the user ratings of android apps. *IEEE Transactions on Software Engineering*, 41(4):384–407, April 2015.

[10] R. V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[11] W. Choi, G. Necula, and K. Sen. Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning. In *OOPSLA'13*, pages 623–640, 2013.

[12] L. Crispin and J. Gregory. *Agile Testing: A Practical Guide for Testers and Agile Teams*. Addison-Wesley Professional, 1 edition, 2009.

[13] A. Dias-Neto, R. Subramanyan, M. Vieira, and G. Travassos. A survey on model-based testing approaches: a systematic review. In *WEASELTech'07*, pages 31–36, 2007.

[14] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. RERAN: Timing- and Touch-Sensitive Record and Replay for Android. In *ICSE'13*, pages 72–81, 2013.

[15] Google. Testing ui for a single app. http://developer.android.com/training/testing/ui-testing/espresso-testing.html.

[16] Google. Ui automator.http://bit.ly/1qIUCTB.

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

[18] M. Grechanik, Q. Xie, and C. Fu. Experimental assessment of manual versus tool-based maintenance of gui-directed test scripts. In *ICSM'09*, pages 9–18.

[19] Y. Hu, T. Azim, and I. Neamtiu. Versatile yet lightweight record-and-replay for android. In *OOPSLA'15*, 2015.

[20] C. S. Jensen, M. R. Prasad, and A. Moller. Automated Testing with Targeted Event Sequence Generation. In *ISSTA'13*, pages 67–77, 2013.

[21] M. E. Joorabchi, A. Mesbah, and P. Kruchten. Real Challenges in Mobile App Development. In *ESEM'13*, pages 15–24, 2013.

[22] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo. Understanding the test automation culture of app developers. In *ICST'15*, pages 1–10, 2015.

[23] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk. Api change and fault proneness: A threat to the success of android apps. In *ESEC/FSE'13*, pages 477–487, 2013.

[24] M. Linares-Vásquez, G. Bavota, M. D. Penta, R. Oliveto, and D. Poshyvanyk. How do API changes trigger Stack Overflow discussions? a study on the android SDK. In *ICPC'14*, pages 83–94, 2014.

[25] M. Linares-Vásquez, C. Vendome, Q. Luo, and D. Poshyvanyk. How developers detect and fix performance bottlenecks in android apps. In *ICSME'15*, pages 352–361, 2015.

[26] 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 *MSR'15*, pages 111–122, 2015.

[27] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An Input Generation System for Android Apps. In *ESEC/FSE'13*, pages 224–234, 2013.

[28] R. Mahmood, N. Mirzaei, and S. Malek. Evodroid: Segmented evolutionary testing of android apps. In *FSE'14*, pages 599–609, 2014.

[29] N. Mirzaei, H. Bagheri, R. Mahmood, and S. Malek. Sig-droid: Automated system input generation for android applications. In *ISSRE'15*, pages 461–471.

[30] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, and D. Poshyvanyk. Auto-completing bug reports for android applications. In *FSE'15*, pages 673–686, 2015.

[31] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk. Automatically discovering, reporting and reproducing android application crashes. In *ICST'16*, page to appear, 2016.

[32] L. Ravindranath, S. nath, J. Padhye, and H. Balakrishnan. Automatic and Scalable Fault Detection for Mobile Applications. In *MobiSys'14*, pages 190–203, 2014.

[33] P. Tonella, R. Tiella, and C. Nguyen. Interpolated N-Grams for Model Based Testing. In *ICSE'14*, 2014.

[34] W. Yang, M. Prasad, and T. Xie. A Grey-box Approach for Automated GUI-Model Generation of Mobile Applications. In *FASE'13*, pages 250–265, 2013.