2017 IEEE International Conference on Software Maintenance and Evolution

How do Developers Test Android Applications?

Mario Linares-Vásquez[1], Carlos Bernal-Cárdenas[2], Kevin Moran[2], and Denys Poshyvanyk[2]
[1]Universidad de los Andes, Bogotá, Colombia
[2]College of William & Mary, Williamsburg, VA, USA
m.linaresv@uniandes.edu.co, {cebernal, kpmoran, denys}@cs.wm.edu

***Abstract*—Enabling fully automated testing of mobile applications has recently become an important topic of study for both researchers and practitioners. A plethora of tools and approaches have been proposed to aid mobile developers both by augmenting manual testing practices and by automating various parts of the testing process. However, current approaches for automated testing fall short in convincing developers about their benefits, leading to a majority of mobile testing being performed manually. With the goal of helping researchers and practitioners – who design approaches supporting mobile testing – to understand developer's needs, we analyzed survey responses from 102 open source contributors to Android projects about their practices when performing testing. The survey focused on questions regarding practices and preferences of developers/testers in-the-wild for (i) designing and generating test cases, (ii) automated testing practices, and (iii) perceptions of quality metrics such as code coverage for determining test quality. Analyzing the information gleaned from this survey, we compile a body of knowledge to help guide researchers and professionals toward tailoring new automated testing approaches to the need of a diverse set of open source developers.**

I. Introduction

Mobile devices have quickly become the most accessible and popular computing devices in the world [2] due to their affordability and intuitive, touch-based user interfaces. The ubiquity of smartphones and tablets has led to sustained developer interest in creating "apps" and releasing them on increasingly competitive marketplaces such as Apple's App Store [18] or Google Play [4]. Due to their highly gesture-driven nature, GUI-based testing of mobile apps is paramount to ensuring proper functionality, performance, and an intelligent user experience. However, GUI-based testing activities are typically costly, and in the context of mobile apps, often performed manually [37], [39]. Given the additional constraints on the mobile application development process including pressure for frequent releases [32], [36], rapid platform evolution and API instability [21], [41], and parallel development across different platforms [13], [37] it can be difficult for developers to budget time for effective testing. Thus, the challenge of automating mobile app testing has captured the interest of the software engineering and systems research communities, and has led to the development different types of automated techniques that assist in various testing tasks.

Research-oriented tools aimed at improving mobile testing span a diverse range, from record & replay approaches [27], [33], to bug reporting aids [51], [52], to automated input generation techniques [1], [3], [15]–[17], [20], [20], [22], [35], [45]–[48], [50], [53], [55], [56], [59]–[61]. Perhaps the most interesting and valuable of these techniques from a developer's or tester's perspective are the automated input generation (AIG) techniques. The high-level goal of such techniques is relatively simple: given a mobile application under test (AUT), generate a series of program *inputs* according to a pre-defined *testing goal*. For the vast majority of these techniques, the generated *inputs* are simulated touch events on the screen of a device, and the *testing goal* is typically either achieving the highest possible code-coverage or uncovering the highest number of faults (e.g., crashes).

However, despite the large amount of research effort dedicated to building AIG techniques and other automated approaches, recent studies seem to indicate that these approaches are typically not used in practice [39]. Choudhary et. al offer a set of potential reasons for this lack of adoption as part of an experience report analyzing several AIG tools [23], and among the reasons reported are (i) lack of reproducible test cases, (ii) side effects across different testing runs, and (iii) lack of debugging support. While this study offers some insight, researchers and practitioners who aim to build these tools with the intention of them gaining adoption and positively impacting developers do not have a clear understanding of the testing needs and preferences of real developers.

This lack of guiding direction for this particular topic of mobile software engineering research is somewhat troubling given the highly practical impact that such tools could have on daily development and testing workflows. Conversely, it is unsurprising that many tools have failed to make an impact without taking into account developer preferences, as *"Automation applied to an inefficient operation will magnify the inefficiency"*[1]. If the *operation* or *goals* of automation techniques do not match developer needs, preferences, and expectations (and are thus *inefficient*), there is little chance that these will have a meaningful impact. Therefore, there is a very clear demand to align the goals of research on automated testing techniques with the needs of developers in order to allow for practical impact.

In this paper, we aim to bridge this gap through a survey, that at its core, aims to examine the testing preferences of open source developers with the intended purpose of providing actionable information to researchers and practitioners working on approaches to automate different aspects of mobile testing.

In summary, this paper makes the following noteworthy contributions:

[1]Bill Gates, co-founder of Microsoft, in reference to automation in business settings

978-1-5386-0992-7/17 $31.00 © 2017 IEEE
DOI 10.1109/ICSME.2017.47

- To the best of our knowledge, this is the first paper aimed at analyzing mobile testing preferences of real open source developers with a focus on (i) typical preferences when designing test cases for mobile apps, (ii) preferred characteristics for automatically generated test cases, and (iii) preferred effectiveness metrics.
- This study complements previous work that has identified and speculated upon potential reasons for lack of adoption of automated mobile testing approaches by collecting information from open source developers and providing a set of learned lessons to guide future research.
- Our general findings indicate that developers (i) rely heavily on usage models of their applications when designing test cases, (ii) prefer high-level, expressive automatically generated test cases organized around use-cases, and (iii) prefer manual testing over automation due to factors including test case representation and issues with reproducibility (iv) do not hold the perception that code coverage is an important measure of test case quality, as indicated by a large portion ($\approx 64\%$) of study participants, instead citing other measures of quality such as feature coverage or fault-detection as more useful.

II. Design of the Empirical Study

The main *goal* of this study is to identify and analyze practices and preferences of mobile developers (MDs) toward testing related activities. To this end, we explored MDs (i) practices in documenting requirements and designing test cases, (ii) preferences toward features of automated testing approaches, (iii) use of existing automated tools, and (iv) preferences for testing-related quality measures. The study is intended to benefit the *perspective* of researchers and practitioners interested in designing approaches and tools for automated testing of mobile apps.

While common wisdom and best practices suggest that test cases should be derived from requirements artifacts, to the best of our knowledge, previous studies focused on the challenges and tools used for testing but without analyzing the details of the strategies used by MDs for designing manual test cases or exploring preferences for automatically generated test cases [37], [39][2]. These aspects are important, as learning the preferences of developers' manual testing practices can inform automated techniques to best meet these needs.

Consequently, we aim to fill this "gap" in recent work by surveying contributors of open source Android apps hosted on GitHub. We are most interested in understanding testing practices of mobile developers from the viewpoint of test case design, preferred testing strategies, reasons for the prevalence of manual testing (as suggested by previous work [37], [39]), and preferred information/features of ideal automatically generated test cases. We also wanted to survey MDs about the usage of widely used techniques in the research community such as random testing and coverage analysis.

[2]Kochhar *et al* [39] investigated also with a survey with 83 open source Android developers the tools they use and challenges they face while testing Android apps. It is worth noting that our survey also includes a question concerning the tools used for automated testing (See **SQ8** in Table I). Kochhar *et al* [39] list a reduced set of tools (i.e., 10), however, we complement their findings with a list of 55 tools used by our surveyed participants. We also complement their findings with a specific question designed to understand experiences and issues of mobile developers when using random testing tools (See **SQ9** in Table I)

TABLE I
Survey Questions for our Empirical Study

Id	Question (Type)
SQ_1	What type of documentation do you use for specifying the requirements in your apps? (Multiple choice)
SQ_2	How do you usually distribute your testing time among these different activities (e.g., manual testing 20%, Junit testing 50%, cloud testing 30%)? (Open question)
SQ_3	Please provide rationale for your answers to SQ3. (Open question)
SQ_4	How do you design the test cases for your apps? (Open question)
SQ_5	What is the target of the test cases you design for your apps? (Multiple choice)
SQ_6	If you are using (or intend to use) tools for automatic generation of test cases, what format for the test cases do you prefer? (Single choice)
SQ_7	Assuming you have test cases in natural language, what type of information would you like to have in them? (Multiple choice)
SQ_8	What tools do you use for automated testing? (Open question)
SQ_9	What are your experiences with random testing tools such as Android Monkey? Are random testing tools useful for your needs? Did you experience any issues with the sequences of events generated by a random testing tool like Android Monkey? (Open question)
SQ_{10}	Do you use code coverage as a metric for measuring the quality of your test cases? Why? (If you answer is No, please describe how else you measure or ensure the quality of your test cases) (Open question)

1) Research Questions: In particular, we aimed at answering the following research questions (RQs):

RQ$_1$: *What are the strategies used by MDs to design test cases?*
RQ$_2$: *What are the MDs' preferences for automatically generated test cases?*
RQ$_3$: *What tools are used by MDs for automated testing?*
RQ$_4$: *Do MDs consider code coverage as a useful metric for evaluating test cases effectiveness?*

2) Data Collection: Table I lists the questions in the survey. SQ_1-SQ_5 were used to answer **RQ**$_1$; SQ_6 and SQ_7 aim at answering **RQ**$_2$; SQ_8 and SQ_9 were designed to answer **RQ**$_3$; and SQ_{10} served to answer **RQ**$_4$. We also collected demographic background information to filter participants with short or over claimed experience in Android development, and to measure the diversity of our sample.

The survey was hosted online on the Qualtrics platform [7], and the participants were contacted via email. To select the potential participants, we followed the same procedure from previous work [44] — that also surveyed open source developers — to extract contributors' emails from GitHub. After the extraction and filtering, we emailed the survey to 10,000 email addresses from which we got 485 survey responses. We discarded 5 responses in which the participant reported 0 years of experience in Android programming, 3 with invalid

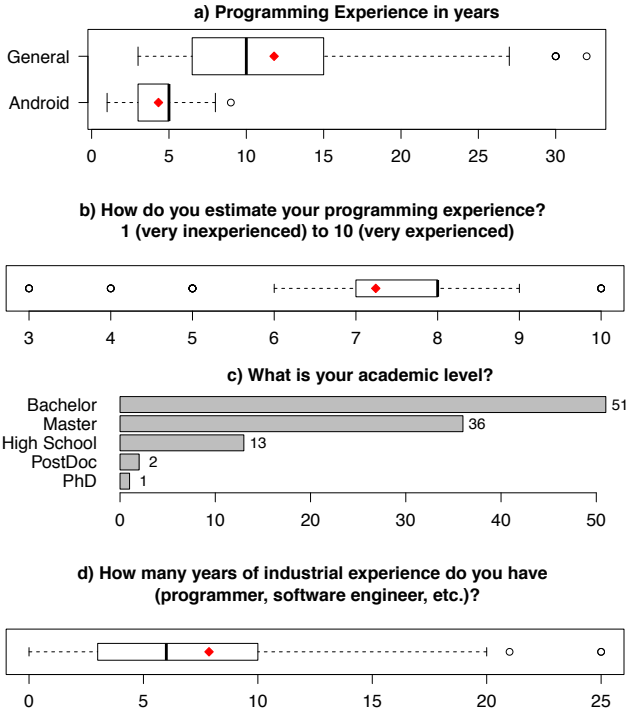

Fig. 1. Results of demographic questions for the 102 survey participants

answers, and 370 unfinished surveys. In the end, we obtained 102 valid responses.

The demographics of the participants are depicted in Figure 1. Our sample (102) is comparable to final numbers of mobile developers (with industrial experience) surveyed/interviewed in previous studies investigating other software engineering phenomenon: 3 in [42], 9 in [49], 45 in [21], 83 in [39], 200 (188 developers + 12 experts) in [37], and 485 in [44]. In addition, the claimed programming experience is diverse for the three cases: general programming, Android programming, and industrial experience.

The answers to multiple/single-choice questions were analyzed using descriptive statistics. In the case of open questions, we categorized the answers manually following a grounded theory-based approach [25]. Three of the authors went through all of the free-text answers and performed one round of open coding by independently creating categories for the answers. After the round of open-coding, the codes were standardized. In the cases of non-agreement between the three coders, corresponding answers were marked as "Unclear".

III. Results and Discussion

In this section we report the responses by the participants and provide answers to the aforementioned research questions. We describe the results using descriptive statistics and through summaries and discussion of examples of free-text answers.

A. RQ_1: What are the strategies used by MDs to design test cases?

Artifacts for documenting app requirements. Android developers use a diverse set of artifacts to document requirements, including artifacts from disciplined and agile methods. Fig. 2 depicts the answers for each of the options in our **SQ1** (note that this was a multiple-choice question). Although there is no large tendency towards a preferred artifact, feature lists are the top-used artifact. Surprisingly, only six participants do not document requirements, and the usage of the other options (i.e., use cases, user stories, and informal natural language descriptions) is balanced across the participants. When analyzing the most popular answers (including combination of choices as a whole answer provided by participants), there is no clear preference; however, we found that the most popular responses reporting the usage of only a single type of artifact as the documentation practice are distributed as follows: user stories (14 participants), feature lists (12), informal natural language descriptions (10), use cases (9). Additionally some participants selected the combination of all 4 artifacts (8) as a single answer.

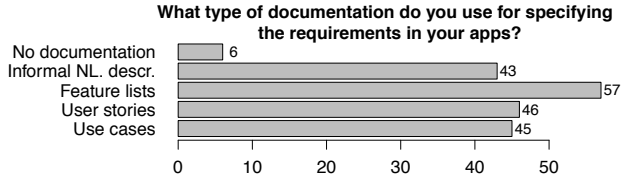

Fig. 2. Artifacts used by Android developers to document apps requirements. The bar plots show the number of times each artifact was selected by the participants.

Distribution of testing efforts. Previous studies have reported that manual testing is preferred over automated approaches [37], [39]. In the survey, we asked participants about how they distribute their testing effort and time across different testing activities (**SQ2**). In particular we asked about the following activities: manual testing, random testing using Monkey, JUnit testing, Record & Replay-based (R&R) testing, GUI ripping-based tools, automated testing with automated testing APIs (ATA), cloud testing services, and others. The answers provided by the participants are depicted with boxplots in Fig. 3.

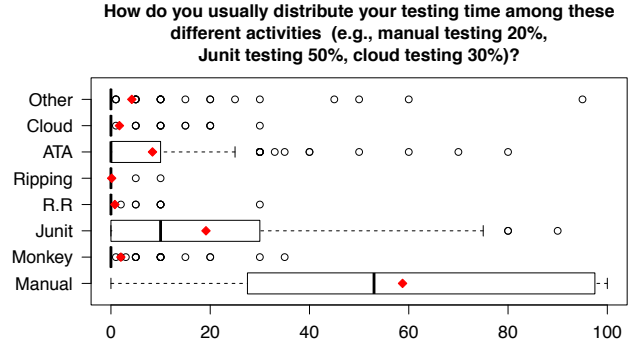

Fig. 3. Distribution of testing activities reported by our participants. *ATA means Automated Testing APIs, and R.R means Record & Replay.

As expected, manual testing is the preferred testing activity with an average of 58.18% of the testing efforts dedicated by the participants, an interquartile range (IQR) of $[25\%, 97.25\%]$ and a maximum of 100%; 96 out of 102 participants reported more than 5% of testing effort devoted to manual testing

with 35 of them reporting more than 90% of dedication to manual testing. The second most popular activity in which the developers dedicate their testing efforts is JUnit-based testing with and average dedication of 18.96%, an IQR of $[0\%, 30\%]$ and a maximum of 90%; 58 participants reported more than 5% of testing effort devoted to unit testing. The third most popular testing activity is automated testing with APIs such as Expresso and Robotium, with an average dedication of 8.29%, an IQR of $[0\%, 10\%]$ and a maximum of 80%; 34 participants reported more than 5% of effort devoted to manual testing.

Fuzz/Random testing with Android Monkey is widely used as the baseline for comparing new automated testing tools proposed by the research community [23], [45], [46], [53]. However, responses from our participants suggest that fuzz/random testing is not widely used in-the-wild. Only 11 participants reported more than 10% of testing effort with Android Monkey with an average of 16.36% (for those 11 participants) and a maximum of 35%. One automated strategy that is widely used in the research community and serves as the foundation for multiple methods of AIG is GUI ripping [15], [16], [20], [54]; however, it seems that developers in-the-wild are not aware of such tools or do not find them useful. Only two participants reported the usage of automated ripping tools with 5% and 10% of their testing efforts. Concerning the case of Record & Replay, five participants reported this technique with more than 5% of testing effort and a maximum of 30%.

Despite of the availability of services such as Xamarin test cloud [58], Saucelabs [11], and Perfecto [6], only 14 participants mentioned the usage of cloud services for testing with an average effort of 12.57% (for those 14 participants) and a maximum of 30%.

Finally, under the "Others" option, we got 14 answers in which the participants claimed more than 10% of testing effort with an average of 29.29% and maximum of 95%. The participants further explained in the answer to **SQ3** that this is mostly because they use customized tools or strategies, instrumentation-based testing, or beta testing with users.

Regarding the rationale provided by the participants for their choices, the preference for manual testing is supported by several reasons such as (i) changing requirements, (ii) lack of time for testing and process decisions, (iii) size of the apps, (iv) lack of knowledge of automated tools and techniques, (v) usability and learning curve of available tools for automated testing, and (vi) the cost of maintaining automated testing artifacts. For instance, the following rationale provided by some participants illustrate their preference for manual testing as a consequence of changing requirements:

> *"our app changes very frequently and we can't afford unit testing and automated testing"*

> *"it's hard to right the useful test case for current project, because the requirement is changed very often, and the schedule is very tiny. So we still prefer hire some tester to do manually testing. In the other hand, the android test framework is not good enough yet, I tried study roboeletric, it's a little bit hard to understand."*

> *"A lot of what I do is related to how the app looks and feels. Therefore a lot of my testing is done manually. When I have complexity in my classes I use junit. Sometimes I use the Instrumentation testing classes from Android, but not much as manually testing feels faster. Also the requirements tend to change a lot so manually testing unfortunately is the best option for me."*

The survey participants also justified the usage of manual testing because of time-related issues and project management decisions. For example:

> *"Usually customers doesn't provide enough time for development of automated tests."*

> *"Too much time required in configuring the components for automated testing. Partly because I was working in a consultancy firm so there was no incentive to spend time on automated testing (not chargable)"*

> *"I do not agree about this method but this is management decision. I repeatedly expressed by disconformity with this methodology."*

> *"Although I strongly disagree with this: the institution I work at does not provide the atmosphere to make testing a vital part of our development."*

> *"My previous work didn't have any testing requirements for the apps, and writing tests takes time that the budget didn't account for. "*

Cost, in terms of money and time, for creating and maintaining automated testing artifacts is also another reason for preferring manual testing. This case is illustrated by the following examples of rationale provided by the survey participants:

> *"I'm faster by testing the app and all it's possibilities on the device itself, instead of writing separate Test Cases."*

> *"Quickly testing the product is important and writing automated tests can't be done quick enough. So a majority of my time is spent making sure the product works manually, then spending time automating what I can. jUnit is wrote by developers, so I've added a couple of tests, but not much. Have just played around with Android Monkey. Our product uses the Cloud, but another person takes care of the majority of the testing."*

> *"We prioritize feature work over automated testing. Automated tests have done little to prevent bugs, but incur significant overhead when creating new features or refactoring old."*

> *"For Android traditionally it has been very difficult to write tests for. Also UI tests can be brittle and take significant effort to maintain. In an environment where development resources are constrained and features / bug fixes works takes priority, it is very difficult to have sufficient tests. Manual testing with a dedicated QA team is more practical to maintain."*

"I'm skeptical of UI testing because I've found that the tests are fragile, require a lot of maintenance, and are generally more work than worth."

The survey participants also claimed a general lack of knowledge of existing automated testing techniques, along with difficulties related to the usability of the tools as factors for not performing automated testing:

"We don't really know how to use other tools."

"I did not know about Android Monkey prior to this survey. I will try it out."

"I have not found any easy-to-use testing solutions for Android"

Finally, the size and maturity of the apps is also a factor that influences the preference for manual testing:

"I never got involved in an Android project big enough to require unit testing, it wasn't worth investing in that."

"I mostly do manual testing due to the limited size of my apps. I sometimes use a custom replay system (built into the app) to duplicate bugs after I come across them. This method is usually combined with manual testing (printing debug information to the log) to pinpoint the cause."

"I'm mostly just building toys or research prototypes, never built Android apps professionally. So I test pretty informally (and poorly) because I just want to build a thing quick and don't care if it's robust."

Test case design strategies. After the open coding for the responses to **SQ4**, 34 answers were not considered because (i) the participants explicitly mentioned they do not perform testing or do not design test cases, and (ii) for some answers we were not able to understand/codify the textual answer. From the valid/accepted answers, the top strategy reported by the participants to design test cases is follow the usage model of the app as a guide (30 answers). The next in the list is designing unit tests for individual components/methods (10 answers), followed by negative testing and edge cases (9 answers), and testing expected outputs (6 answers). Bugs, changes in the last version, and regression account for 9 answers. Three participants claimed they follow the Behavior-Driven-Development philosophy (BDD). In addition, three participants mentioned they perform ad-hoc testing. Finally, two of the participants combine the usage model with feedback from the end users.

Non-functional requirements were mentioned only in few cases as the drivers for designing test cases: robustness (2 answers), performance (1 answer), usability (1 answer), and different device configurations (1 answer). Other strategies mentioned by only one participant each are: code coverage, defining assertions in code, dependencies, testing the GUI model, and testing the business logic.

We also found that developers tend to prefer a single criteria for designing test cases, as very few respondents reported more than one preference. Only 15 participants reported mixed strategies, as those described in the following responses:

"l look for boundary conditions - i try to work out what happens in the Grey Areas - i look for ways of breaking it. i also test a range of use cases, how Can the user interact with the app? when combinations are possible? i try to test the most probable Scenarios and some strange ones."

"1) specific tests depending on what the app should do. For instance: schedule should be as precise as is required for a scheduling app. 2) robustness: find the limits on the app by constantly changing the aspect ratio of the screen or switching app on and off. 3) look at CPU utilization while testing the app... app should not drain battery unnecessary"

"Based on the user stories or Use cases. Define initial state (local data). Perform scenario (call a rest service / perform an activity, etc). Validate final screen or rest service result"

The answers to **SQ4** were complemented by participants reporting the testing goals they have when designing test cases (**SQ5**). Fig. 4 depicts the responses for **SQ5**. As shown in Fig. 4, the developers prefer to design test cases that target individual uses cases/features (77 answers), or combinations of multiple uses cases/features (49 answers). Random events were mentioned only by 16 participants. The option "Other" was selected by nine participants; four of the "Other" answers were "none", two participants mentioned non-functional attributes (i.e., performance and robustness), one participant responded "corner cases", and one mentioned "unit test".

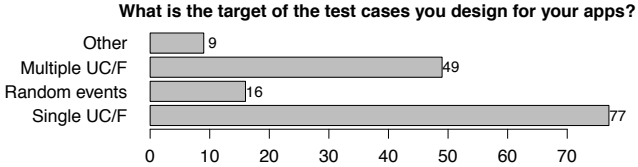

Fig. 4. Target of test cases designed by the survey participants. The bar plots show the number of times each target was selected by the participants. *UC means Use Case, and F means Feature.

Answer to RQ$_1$. Mobile developers (as represented by our survey sample) heavily rely on usage models to document and design test cases. First, requirements are documented using different artifacts such as feature lists, informal natural language descriptions, user stories and use cases; only six out of 102 participants reported they do not use any artifact to document requirements. Second, the surveyed participants mostly rely on manual testing and unit testing for their testing strategies. The rationale provided by the participants for their preference and effort dedication to manual testing is supported by several reasons such as (i) changing requirements, (ii) lack of time for testing and process decisions, (iii) size of the apps, (iv) lack of knowledge of the tools and techniques, (v) usability and learning curve of available tools, and (vi) the cost of maintaining automated testing artifacts. Finally, the surveyed developers mostly focus on the usage model to design test cases, and use one or a combination of use cases/features as the target for their test cases.

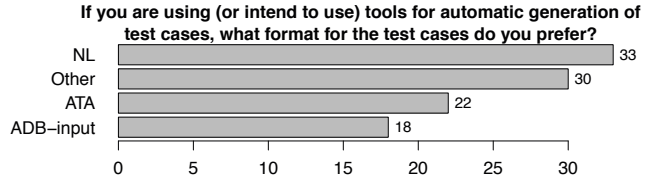

Fig. 5. Automatically generated test cases format preferred by the survey participants. NL means Natural Language, ATA means Automated Test API, and ADB-input means input commands generated via the Android Debug Bridge (ADB).

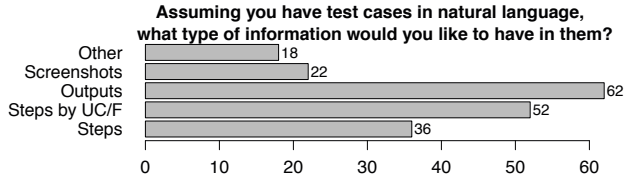

Fig. 6. Information preferred by survey participants in automatically generated test cases. UC means Use Case and F means Feature.

B. RQ_2: What are MDs preferences for automatically generated test cases?

Concerning **SQ6**, natural language is the format preferred by the survey participants (See Figure 5) for automated test cases, with 33 out of 102 participants selecting this option. The second most popular answer was the "Other" option (30 participants), which represented mostly the lack of participant knowledge about tools for automatic generation of test cases or the lack of preference for any format. Unfortunately, the option "Other" did not provide us with actionable knowledge; 17 textual answers for the "Other" option are empty or claim no preference for any format; ten participants reported that they do not use/do not like/are not aware of tools for automatic generation of test cases; one answer was unclear; one answer mentioned scripts augmented with comments; and one participant responded "Replayable event streams, starting from a known good state".

The third most popular option for **SQ6** was test cases written with automated testing APIs such as Espresso and Robotium (22 participants), and the least selected choice was ADB input commands (18 participants). Therefore by combining the number of participants voting for natural language test cases and test cases written with ATAs, the results suggest that Android developers prefer expressive test cases over low level scripts with input commands.

Concerning the preferred information that developers would like to have in an ideal automatically generated test case (**SQ7**), expected outputs and reproduction steps organized by use case or feature are preferred over the other options. The answers for **SQ7** are depicted in Fig. 6. 82 out of 102 participants (80.39%) selected either "Expected outputs" or "Reproduction steps grouped by use case/feature". Only 21.7% of the participants agreed on having screenshots as part of test cases. And, the textual answers to the "Other" option include: reason/motivation for the test case, device and contextual information (e.g., Android OS version, display dimension, internet connection status), malicious user inputs, and specifications like in the RSpec framework for Ruby [10].

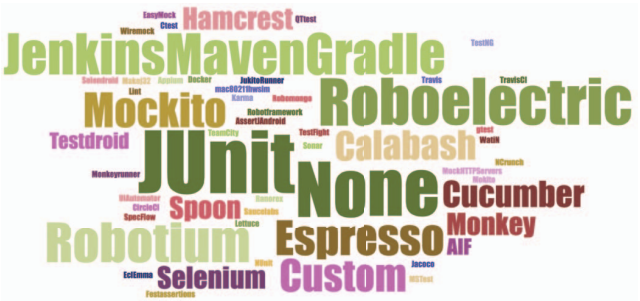

Fig. 7. Tools used by the participants for automated testing of Android apps.

Answer to RQ_2. Automatically generated test cases in natural language or expressed using automated testing APIs (e.g., Robotium or Espresso) are preferred by the participants. This suggest a preference for high-level languages instead of low level events (e.g., using ADB input commands). In addition, the surveyed developers prefer to have test cases that include expected outputs and reproduction steps organized/grouped by use cases/features.

C. RQ_3: What tools are used by MDs for automated testing?

Tools for automated testing. 55 different tools have been used by our survey participants (**SQ8**); the tools and frequencies are depicted as a word cloud in Fig. 7. The most used tool is JUnit [5] (45 participants), followed by Roboelectic [8] with 16 answers, and Robotium [9] with 11 answers. 28 participants explicitly mentioned they have not used any automated tool for testing mobile apps. 39 out of 55 tools were mentioned only by one participant each, which suggests that mobile developers do not use a well established set of tools for automated testing. In addition, surprisingly, Monkey [31], the state-of-the-art tool for fuzz/random testing, was mentioned only by three out of 102 participants, and the results are similar for other tools designed/promoted by Google: Espresso [30] (eight participants), MonkeyRunner [29](one participant), Lint [28] (one participant), and UIAutomator [30] (one participant). None of the mentioned tools allow for automatic derivation of test cases from source code, real app usages, or requirements specifications[3]

Experiences with random testing tools. 60.78% (62 out of 102) of the participants reported no experience with Monkey or tools for random testing (**SQ9**). 15 participants (14.71%) provided no-rationale/unclear answers. Concerning usefulness, 13 participants (12.75%) consider Monkey and random testing as useful tools for corner cases or stress testing, and in some cases for finding performance issues. For instance:

> *"Yes, it's useful to detect minor/stability issues. For instance it sometimes finds issues happened when tapping buttons many times quickly."*

> *"Monkey is very useful for stress testing the application or to verify that there are no leaks (typically memory*

[3]It is worth nothing that the Android Monkey tool generates random sequences of events, however, the sequences are not easy to document or describe in terms of use cases.

leaks) that build up over time. Sometimes, it also catches the odd bug as well."

However, eight of those 15 participants pointed out that some issues or limitations are related to low impact of the discovered bugs and reproducibility:

"They can be useful in finding memory leaks, ANRs, bad navigation flows and the like. They can be problematic in doing unexpected things, e.g. exiting your application during a test run. "

"Good for stress testing, not very consistent results."

"They add a lot of noise for very little signal. They are good at finding some really weird cases, but mostly it seems like the cases they find wouldn't normally be hit by a user and that time could be better spent adding features."

"I used it long time ago and find that it works as a solution only for low-quality fragile code. It's rarely helps to improve overall quality of the app."

11 participants (10.78%) provided answers in which the main message is that Monkey/Random testing tools are not useful because of reproducibility issues, maintenance costs of the scripts, and lack of tangible benefits:

"hard to reproduce bugs, steps hardly reproducible by human beings"

"I've ran Android Monkey and it found some defects, but the developers said "that barely happens" or "that never happens". So the defects weren't looked into."

"I don't think random tests provide any value. The ideal is to have a tool able to perform an entire session (run several scenarios) / without having the test-runner to kill and initialize the app before running each scenario."

Although random testing has been proven to be "useful" by the research community [14], [19], [24], [26], available random testing tools for mobile apps (e.g., Monkey) have issues such as lack of expressiveness. For instance, the Android Monkey tool allows for reproducibility of event streams (by using the same seed), but it does not have log capabilities for creating a higher level representation of the streams. In addition, it seems to be the only tool available for the community for random testing of mobile apps.

Researchers have also designed tools for mobile random testing (e.g., Dynodroid [46]), however those tools are still only known by the research community or have impediments for industrial usage (e.g., applicability only under certain conditions). In general, the main finding here is the lack of experience with and lack of usage of random testing tools by the surveyed participants. Regarding the usefulness, few participants reported cases in which the tools are useful for finding corner cases and stress testing; and few participants reported that the tools are not useful at all. However, common complaints in both cases (from both participants who saw benefits and those who didn't) are lack of reproducibility of the event sequences. These results suggest, that current tools used by Android developers need to be improved to allow expressiveness of the generated streams. Also more effort in the research community should be dedicated to promote the usage of random testing tools generated as part of research, and to deliver tools that can be easily adopted by the industry.

Answer to RQ_3. The surveyed participants rely on a diverse set of tools for supporting automated testing of mobile apps. In particular, 55 different tools were reported showing a preference for APIs such as JUnit, Roboelectric and Robotium. Compared to the study by Kochhar *et al.* [39], we report a larger set of tools answered by a larger set of participants. However, both studies agree on listing JUnit, Roboelectric and Robotium as part of the top-4 used tools. Record & replay, and random testing tools are used only by few participants. In the case of random testing tools, few participants claimed some benefits such as stress testing, execution/discovery of corner cases, and execution of events that are hard to generate by humans. However, impediments for increased adoption of random testing tools (e.g., the Android Monkey tool) are the lack of expressiveness of the generated event streams, and difficulty reproducing scenarios.

D. RQ_4: Do MDs consider code coverage as a useful metric for evaluating test cases quality?

14 out of 102 participants reported they do not use code coverage, do not use automation tools, or were unaware of code-coverage as a quality metric (**SQ10**); and six out of 102 participants provided no valid answers. From the remaining 82 participants, 51 answered "No" (i.e., code coverage is not useful), 29 answered "Yes" (i.e., code coverage is useful), and two participants provided a "yes-no" answer. In the case of the "No" answers, 19 out of 51 augmented the answer claiming that code coverage is not a good metric for measuring quality of test cases because there are other useful and better methods/metrics such as code (test cases) reviews, number of faults detected by the test cases (fault-detection capability), features covered by the test cases (feature coverage), or the "works for me" criteria[4]. Examples of the answers claiming that code coverage is not a useful metric for evaluating test case quality include the following:

"No. We measure the number of uncaught bugs and regressions over time that devs had to spend time fixing"

"No, calculate total coverage based on features, covered elements etc."

"I don't usually participate in the testing side of things, but I wouldn't use code coverage as a metric for quality as the two are completely distinct and different things."

"Code coverage categorically does not measure the quality of tests. It is useful to show that code is not

[4]Note that the effectiveness of code coverage for measuring the quality of test suites has been already questioned by the software engineering community [34], [38], [62].

currently tested but it says nothing further about the code that is already under test. Many people – probably most – are quite skilled at writing useless tests. Education is the only tool for producing high-quality tests and code review is the only tool for ensuring that quality. That said, fuzz testing, for instance, is a very powerful tool for certain kinds of testing. Knowing how to use the tools available to us is part of that education."

29 out of 102 participants found code coverage a useful metric for measuring test cases quality, in particular for identifying code entities that have not been tested. Examples of their answers are as in the following:

"I use code coverage mainly as a tool to ensure I haven't forgotten any major areas of testing. Most of my projects have a minimum coverage requirement of 75-80%"

"Yes, I try to keep code coverage at an acceptable level. This is definitely not the only thing that matters, but I think it does matter."

"We use code coverage because it's easy to measure, it's a good enough metric, and because if developers feel they are being measured they are more likely to write more tests, thus generally producing the desired outcome."

Finally, the "Yes-no" answers claim code coverage is not useful at all, but they help to identify parts of the code that have not been tested:

"Yes and no. It's an indication if something is tested, not that the test is correct."

"No. We use code coverage more as a guide to which part of the code base might need more attention in terms of writing more tests. We don't really have other metric for measuring quality of the test cases."

Answer to RQ$_4$. Code coverage is not used or not considered as useful for measuring the quality of test cases by 63.73% (i.e., 65 out of 102) of the surveyed participants. Some of the reasons explaining the lack of confidence in the metric is that they prefer fault-detection or feature-coverage capabilities of test cases as a measure of quality, or they prefer to measure test cases quality by performing code reviews. On the other side, some participants consider that code measure is a useful metric because it helps to identify parts of the code that are not tested.

IV. Threats to Validity

Threats to *construct validity* concern the relationship between theory and observation, and relate to possible measurement imprecision when extracting data used in a study. To minimize this threat we filtered out incomplete surveys, participants with zero years of experience, and surveys with invalid answers. Moreover, to minimize a source of inexactitude in our study on the open questions, we followed a grounded theory-based approach [25]. In particular three of the authors performed an open coding by independently creating categories, then the codes were standardized and in case of no-agreement the answers were marked as "Unclear".

Threats to *external validity* concern the generalizability of our findings. The results in our study may not be generalizable to developers on other platforms, moreover our study only focuses on developers from open source projects on Github and we can not guarantee that all participants are commercial developers (although the average industrial experience for participants is between 5-10 years). In addition, the testing practices and automated tools used by our sample set may not generalize across all mobile developers. However, despite this fact, we believe the information provided by our respondent pool can be used to effectively provide guidelines for the research community towards devising more practical automated testing approaches.

Another threat related to generalizability is that the results of our study are based on 102 respondents which might not be representative of the global community of Android developers. However, this study surveys a comparable number of Android developers to other studies [12], [37], [39].

V. Related Work

In this section we present the related work and we differentiate the outcomes of our study compared to other studies concerned with investigating the topic of mobile testing.

Erfani *et al.* [37] performed a study to understand the challenges that developers face during the life cycle of mobile software development. The study comprises interviews and a semi-structured survey targeted to 12 experts on mobile development and 188 people from the general mobile community respectively. Therefore, the findings can be categorized in four main topics: (i) general challenges such as fragmentation, testing support, open/closed platforms, data intensive, and frequent code changes; (ii) development across multiple platforms with problems such as native vs. hybrid apps, capabilities of platforms, code reuse vs. writing from scratch, behavioral consistency cross platform, and effort on migration across platform; (iii) current testing practices like manual testing, developers as testers, platform specific testing, levels of testing, beta testers; and finally (iv) testing challenges including limited unit test support for mobile specific features, better monitoring support, crash reports, emulators, missing platform-supported tools, rapid changes, multiple scenarios to validate, app stores and usability testing.

The study concluded that one of the most important challenges for developers is having to deal with multiple platforms, since the knowledge of one platform typically can't be transferred to another. In addition, tools to monitor and measure the performance of mobile apps are important for developers as are testing frameworks and tools. Our paper differs from the goals of this study in that we focus on examining the testing practices and preferences of open source developers whereas Erfani *et al.* analyzed a variety of aspects of the entire software development process.

Kochhar *et al.* [39] conducted an empirical investigation into open source apps and two different surveys, the first one com-

prising three questions asked to 83 android developers, and the second comprising five questions as an improved version of the first study, posed to 127 windows app developers at Microsoft. The study includes questions to investigate techniques used to test apps, frameworks used, types of testing used, reasons for using testing tools, and challenges encountered during testing process. The authors concluded that Android apps are not properly tested since around 86% of the apps do not contain any test cases. In addition, existing automated tools are not able to reach certain parts of code in mobile apps and are typically prohibitively difficult to use. Finally, the study found that developers are not aware of many existing testing tools. Our paper differentiates itself in the fact that we attempted to distill developer's testing preferences and practices for both manual and automated practices in order to inform the development of more practical automated tools, and our participants were Android developers.

Choudhary *et al.* [23] presented a comparison between test input generation techniques for Android applications. Choudhary et al. studied these tools applied to 60 real-world applications considering four different criteria: (i) ease of use, (ii) android framework compatibility, (iii) code coverage achieved, and (iv) fault detection. The authors concluded that random testing (specifically Android Monkey) surpasses all other automated techniques. In contrast to this study we surveyed developers to investigate trends on usage of automated testing tools and experiences with random testing tools.

Linares-Vásquez *et al.* [43], recently conducted a survey of current tools, frameworks, and services available to support mobile testing practices. This survey draws comparisons between different testing techniques and solutions, describing the benefits, and delineating drawbacks and trade-offs between different approaches/tools. Additionally, the work offers a forward-thinking vision for effective mobile testing along three principles: Continuous, Evolutionary, and Large-Scale. While this work offers a valuable perspective on the current state and potential future of mobile app testing, it does not survey developers to understand current mobile testing trends.

Aho *et al.* [12], presented an industrial evaluation of the *Murphy tool* that models the graphical user interface to support several testing tasks during the software development cycle. The experiences presented in the paper were based on the evaluation of three software systems and three test engineers from industry. The Murphy tool decreased the time and effort of generating test cases from the model. The authors concluded that Murphy helped to minimize the tedious and repetitive work while creating manual test cases that involves analysis and verification from the tester. Compared to this study, we do not focus on the evaluation of one particular approach rather we surveyed open source developers about the usage of different automated testing tools, and preferences for ideal automated testing techniques.

VI. Conclusion and Learned Lessons

In this paper we presented the results of an empirical study with 102 contributors of open source mobile apps hosted at GitHub. In particular, the study was conducted with a survey aimed at gathering information about their practices in-the-wild and preferences for (i) designing and generating test cases, (ii) using automated approaches, and (iii) assessing the quality of test suites.

Our survey reveals highly relevant opinions of open source developers such as they rely primarily on usage models (e.g, use cases, user stories) of their applications when designing test cases, and they prefer high-level expressive automatically generated test cases organized around use-cases. As of today, little effort has been devoted to include usage models [40], [45], [57] during automated test cases generation for mobile apps; thus, usage models and expressive test cases should be considered as an important goal for automated approaches/tools for mobile testing. The survey results support the need for multi-models in model-based testing as suggested by Linares-Vásquez *et al.* [43].

Another result we would like to highlight is the fact that code coverage is not perceived by the survey participants as an important measure of test cases quality. While code coverage has been widely used by researchers to validate automated approaches for testing mobile apps, this result could be used as an insight that reinforce the discussion regarding code coverage utilty [34], [38]. Additionally, this should spur the discussion and creation of new evaluation models — for new testing approaches/tools — that consider other criteria such as relevant fault detection capability (e.g. faults along heavily traversed parts of the app) and feature coverage.

Finally, our survey confirms the fact that despite the plethora of tools proposed by the research community, the state-of-the-practice for automated testing are automation APIs; manually written test cases with automation APIs are very fragile to changes in the GUI of the app under test test [37], [43]; even the official Google tool for random/fuzz testing (i.e., Monkey) has a low usage rate. In order to aid in technology transfer, researchers should consider developer preferences and workflows when designing and evaluating their approaches. Such preferences can be gleaned from the developer responses in this paper, and include among others: (i) A need for automatically generated test cases to co-evolve with apps and features, (ii) low-overhead tools that tightly integrate into current (agile) development workflows, and (iii) expressive test cases that allow for easier debugging and traceability between test cases and features. By taking such preferences into consideration, researchers should be able to design approaches that make a meaningful impact during real mobile testing practices.

References

[1] Android ui/application exerciser monkey http://developer.android.com/tools/help/monkey.html.
[2] Current number of smartphones in use https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/.
[3] Google firebase test lab robo test https://firebase.google.com/docs/test-lab/robo-ux-test.
[4] Google play store https://play.google.com/store?hl=en.
[5] Junit. http://junit.org.
[6] Perfecto. http://www.perfectomobile.com.
[7] Qualtrics. http://www.qualtrics.com.
[8] Roboelectric. http://robolectric.org.

[9] Robotium. https://code.google.com/p/robotium/.

[10] Rspec. http://rspec.info.

[11] Sauce labs. https://saucelabs.com/features/#features-automated-mobile.

[12] 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.

[13] M. Ali, M. E. Joorabchi, and A. Mesbah. Same app, different app stores: A comparative study. In *MOBILESoft'17*, pages 79–90, Piscataway, NJ, USA, 2017. IEEE Press.

[14] D. Amalfitano, N. Amatucci, A. R. Fasolino, P. Tramontana, E. Kowalczyk, and A. Memon. Exploiting the saturation effect in automatic random testing of android applications. In *MOBILESoft 2015*, pages 33–43, 2015.

[15] 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.

[16] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. Memon. Mobiguitar - a tool for automated model-based testing of mobile apps. *IEEE Software*, page to appear, 2014.

[17] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *FSE'12*, 2012.

[18] Apple. App store. https://itunes.apple.com/us/genre/ios/id36?mt=8, 2017.

[19] A. Arcuri, M. Z. Iqbal, and L. Briand. Random testing: Theoretical results and practical implications. *IEEE Trans. Softw. Eng.*, 38(2):258–277, Mar. 2012.

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

[21] G. Bavota, M. Linares-Vásquez, C. Bernal-Cárdenas, M. Di 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 (TSE)*, 2015.

[22] 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.

[23] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for android:are we there yet? In *29th IEEE/ACM International Conference on Automated Software Engineering (ASE'15)*, 2015.

[24] I. Ciupa, A. Pretschner, M. Oriol, A. Leitner, and B. Meyer. On the number and nature of faults found by random testing. *Softw. Test. Verif. Reliab.*, 21(1):3–28, Mar. 2011.

[25] J. Corbin and A. Strauss. Grounded theory research: Procedures, canons, and evaluative criteria. *Qualitative Sociology*, 13(1):3–21, 1990.

[26] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *IEEE Trans. Softw. Eng.*, 10(4):438–444, July 1984.

[27] 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.

[28] Google. Lint. http://developer.android.com/tools/help/lint.html.

[29] Google. monkeyrunner. http://developer.android.com/tools/help/monkeyrunner_concepts.html.

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

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

[32] G. Hu, X. Yuan, Y. Tang, and J. Yang. Efficiently, effectively detecting mobile app bugs with appdoctor. In *Ninth European Conference on Computer Systems (EuroSys'14)*, page Article No.18, 2014.

[33] Y. Hu, T. Azim, and I. Neamtiu. Versatile yet lightweight record-and-replay for android. In *OOPSLA'15*, OOPSLA 2015, pages 349–366, New York, NY, USA, 2015. ACM.

[34] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 435–445, New York, NY, USA, 2014. ACM.

[35] 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.

[36] N. Jones. Seven best practices for optimizing mobile testing efforts. Technical Report G00248240, Gartner, 2013.

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

[38] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 654–665, 2014.

[39] 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.

[40] E. Kowalczyk and A. Memon. Extending manual gui testing beyond defects by building mental models of software behavior. In *ASEW'15*, pages 35–41, Washington, DC, USA, 2015. IEEE Computer Society.

[41] 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.

[42] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk. Optimizing energy consumption of guis in android apps: A multi-objective approach. In *ESEC/FSE'15*, pages 143–154, 2015.

[43] M. Linares-Vásquez, K. Moran, and D. Poshyvanyk. Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. In *33rd IEEE International Conference on Software Maintenance and Evolution (ICSME'17)*, page to appear, 2017.

[44] M. Linares-Vásquez, C. Vendome, Q. Luo, and D. Poshyvanyk. How developers detect and fix performance bottlenecks in android apps. In *ICSME'15*, page to appear, 2015.

[45] 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*, 2015.

[46] 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.

[47] R. Mahmood, N. Mirzaei, and S. Malek. Evodroid: Segmented evolutionary testing of android apps. In *FSE'14*, page to appear, 2014.

[48] K. Mao, M. Harman, and Y. Jia. Sapienz: Multi-objective automated testing for android applications. In *ISSTA'16*, pages 94–105, 2016.

[49] M. Miranda, R. Ferreira, C. R. B. de Souza, F. Figueira Filho, and L. Singer. An exploratory study of the adoption of mobile development platforms by software engineers. In *MOBILESoft'14*, pages 50–53, New York, NY, USA, 2014. ACM.

[50] N. Mirzaei, H. Bagheri, R. Mahmood, and S. Malek. Sig-droid: Automated system input generation for android applications. In *ISSRE'15*, pages 461–471, Nov 2015.

[51] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, and D. Poshyvanyk. Auto-completing bug reports for android applications. In *FSE'15*, page to appear, 2015.

[52] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, and D. Poshyvanyk. Fusion: A tool for facilitating and augmenting android bug reporting. In *ICSE'16*, May 2016.

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

[54] B. Nguyen and A. Memon. An observe-model-exercise* paradigm to test event-driven systems with undetermined input spaces. *IEEE Transactions on Software Engineering*, 99(Preprints), 2014.

[55] 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.

[56] R. Sasnauskas and J. Regehr. Intent fuzzer: Crafting intents of death. In *WODA+PERTEA'14*, pages 1–5, 2014.

[57] P. Tonella, R. Tiella, and C. Nguyen. Interpolated n-grams for model based testing. In *International Conference on Software Engineering (ICSE'14)*, 2014.

[58] Xamarin Inc. Xamarin test cloud. https://xamarin.com/test-cloud.

[59] 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.

[60] R. N. Zaeem, M. R. Prasad, and S. Khurshid. Automated generation of oracles for testing user-interaction features of mobile apps. In *ICST'14*, pages 183–192, Washington, DC, USA, 2014. IEEE Computer Society.

[61] H. Zhang and A. Rountev. Analysis and testing of notifications in android wear applications. In *ICSE'17*, May 2017.

[62] Y. Zhang and A. Mesbah. Assertions are strongly correlated with test suite effectiveness. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 214–224, New York, NY, USA, 2015. ACM.