Random GUI Testing of Android Application Using Behavioral Model

Woramet Muangsiri*, Shingo Takada*
*Dept. of Information and Computer Science, Keio University, Yokohama, Japan
{w.muangsiri, michigan}@doi.ics.keio.ac.jp

***Abstract*—Automated GUI testing based on behavioral model is one of the most efficient testing approaches. By mining user usage, test scenarios can be generated based on statistical models such as Markov chain. However, these works require static analysis before starting the exploration which requires too much prerequisites and time. In this work, we propose a behavioral-based GUI testing approach for mobile applications that achieves faster and higher coverage. Our approach does not conduct static analysis. It creates a behavioral model from usage logs by applying a statistical model. The events within the behavioral model is mapped to GUI components in a GUI tree. Finally, it updates the model dynamically to increase the probability of an event that rarely or never occurs when users use the application. We evaluated our approach on three open-source Android applications, and compared it with other approaches. Our approach showed the effectiveness of our tool.**

Keywords- Testing Tools; Testing Automation; GUI Testing; Android; Behavioral Model;

I. INTRODUCTION

It is undeniable that the smartphone has become a part of our life, and adoption has reached almost 3 billion active smartphone users in 2016 and still increasing year by year. Due to the huge and intensely competitive market, the high-quality and flawless graphic user interface is an essential part of success. Moreover, the mobile application design trend has been evolving rapidly. GUI testing is a very important part of achieving a high-quality application.

Various methods for automated GUI testing have already been studied, implemented, and evaluated [1]. However, according to a study conducted by Joorabchi et al [2], more than 70% of the survey respondents preferred to adopt manual GUI testing in practice, and less than 5% are engaged in fully automated GUI testing. Moreover, more than half of the participants admitted that GUI testing is challenging to automate and is still labor intensive.

To overcome these challenges, our goal is to create an automated tool that can be adopted easily, with smaller dependency and higher efficiency. We propose a behavioral-based automated GUI testing approach which requires as little setup effort as possible, while still achieving equivalent or better code coverage than other automated approaches by:

1) Avoiding static analysis by mapping the GUI tree and behavior model during runtime.
2) Updating the model dynamically to increase the probability of an event that rarely or never occurs when the user uses the application.
3) Applying a statistical model to create a behavioral model.

The remainder of this paper is organized as follows: Section 2 provides related works on GUI testing, Section 3 describes the methodology as well as the architecture, Section 4 evaluates our approach, Section 5 outlines the limitations of our tool, and Section 6 contains our conclusions and future works.

II. BACKGROUND AND RELATED WORKS

There are a number of works related to automated GUI testing for the Android platform. One of these is a tool called Monkey [3] which is bundled with Android SDK and is the most frequently used tool to test Android apps.

Android Monkey generates and sends pseudo-random streams of GUI events to the *Application Under Test* (AUT). It is fully automatic, and it can efficiently test the AUT with a large number of simple inputs. However, it is not suited for generating inputs that require human intelligence, and it tends to generate redundant inputs.

Other approaches such as model-based approaches [4], [5] have been proposed. Dynodroid [5] is based on random exploration with several strategies that make its exploration more efficient compared to a uniform distribution random testing. Frequency strategy selects the events that have been least frequently selected so far. Biased random strategy, like the frequency strategy, maintains a history of how often each event has been selected, but in a context-sensitive manner.

The model-based exploration strategies consider each individual activity as a state, and each event as a possible transition. Amalfitano et al. [4] presented *AndroidRipper*, an automated GUI-based tool to test Android apps in a structured manner. It implements a depth-first search (DFS) strategy and restarts the exploration from a fresh starting state when it cannot go to any other states during the exploration.

Recently, several techniques [6], [7], [8], [9] which captures user's input have been proposed. Linares-Vasquez, et al. [8] created a tool called *MonkeyLab* which mines GUI-based models that are used to generate actionable scenarios for both natural and unnatural sequences of events. Gomez et al. [9] proposed a crowdsourced approach and applied Path Analysis and Sequential Patterns algorithms to reproduce context-sensitive crashes by real users.

DOI reference number: 10.18293/SEKE2017-099

III. METHODOLOGY

Although the basis of our approach of using a behavioral model is not new, our approach is unique in that we continually update the behavioral model as the test progresses. The initial behavioral model is created from usage logs using statistical modeling, but we also consider events that did not appear in the usage logs. This is done through mapping events in the behavioral model with GUI components in the GUI tree. Event probabilities are adjusted as the test progresses.

Fig. 1 shows the overall architecture of the system. In this paper, the photo editing application in Fig. 2 will be used as a running example. Our approach can be divided into six steps as follows:

1) Gather the usage logs beforehand and extract GUI tree from the AUT (Section III-A).
2) Apply a statistical model to create a behavioral model, then map with GUI Tree (Section III-B).
3) Select an event from the behavioral model randomly (Section III-C).
4) Fire the selected event to the AUT (Section III-D).
5) Adjust the probability by frequency and update the model if needed (Section III-E).
6) Repeat from (2) to (5) until reaching a time limit or pre-decided number of events.

A. Observation

The purpose of this step is to acquire the mandatory components, the usage logs and GUI tree, for generating behavioral model.

1) Obtain Usage logs: This is the only prerequisite step before runtime. Usage logs are obtained from the AUT by using *Recorder*, the usage recording service. The *Recorder* service was implemented using Android's *AccessibilityService*[1] which captures events executed by the testers or users in the form of *AccessibilityEvent*. These events are then collected and transferred to the *Observer* through *Android Debug Bridge* (ADB). We focus on the events shown in Table I. Once the user starts using the application while the *Recorder* service is turned on, a recording session starts, which will end when the user closes the application. Since this data is the basis for the behavioral model, it is better to have many sessions. One way to obtain a large amount of sessions is to crowdsource [9]. This process does not require the developer to modify AUTs source code.

2) Extract GUI Tree from AUT: The GUI tree is a hierarchy of GUI components, extracted from the AUT during runtime using Android's *Hierarchy Viewer*[2]. We chose this dynamic approach, rather than statically analyzing the AUT's apk file; this results in adding less than $100ms$ overhead per action. Fig. 3 shows a simplified GUI tree from the example activity in Fig. 2. In this example, the simplified GUI Tree includes 6 GUI components, e_1 *backButton,* e_2 *editButton,* e_3 *deleteButton,* e_4 *shareButton,* e_5 *imageView, and* e_6 *saveButton*. In the GUI tree, each component contains its *type, resource-id, text, coordinate,* and *possible actions* that the user can take.

[1]https://developer.android.com/training/accessibility/service.html
[2]https://developer.android.com/studio/profile/hierarchy-viewer.html

TABLE I
ACCESSIBILITY EVENT TYPES

Event type	Description
TYPE_VIEW_CLICKED	Represents the event of clicking on a *View*.
TYPE_VIEW_LONG_CLICKED	Represents the event of long clicking on a *View*.
TYPE_VIEW_SCROLLED	Represents the event of scrolling on a *View*.
TYPE_VIEW_TEXT_CHANGED	Represents the event of changing the text of an *editText*.
TYPE_WINDOW_STATE_CHANGED	Represents the event of opening a *PopupWindow, Menu, Dialog, etc*.

B. Behavioral Model Generation and Mapping

In this step, the behavioral model is created from usage logs, then events are mapped to GUI components in the GUI tree.

1) Behavioral Model Generation: The behavioral model is created from usage logs by approximating the conditional probabilities using the *Markov assumption* (see a simplified behavioral model in Fig. 4). We use 5-grams model. An n-gram model is simply a sequence of words. In the context of software testing, these words are events. The *n* refers to the number of events. Consider a sequence of events $e_{tap1}, e_{tap2}, e_{hold1}, e_{swipeUp}$. If $n = 2$, then the 2-gram or *bigram* would be:

1) $p(e_{tap2}|e_{tap1})$
2) $p(e_{hold1}|e_{tap2})$
3) $p(e_{swipeUp}|e_{hold1})$

And for $n = 3$, the *3-grams* or *trigram* would be:

1) $p(e_{hold1}|[e_{tap1}, e_{tap2}])$
2) $p(e_{swipeUp}|[e_{tap2}, e_{hold1}])$

Where $p(e_x|[e_j, e_k])$ is the probability of event e_x being selected given history events $[e_j, e_k]$. With larger n, a model can store more context with a well-understood space-time tradeoff. It is possible to use higher than 5-gram if more sessions of usage log and processing power are available.

In practice, the app might not be entirely explored by users. In Fig. 4, suppose that there is another event e_6 that can occur when the user is at Activity B, but it is missing from the simplified behavioral model. This means that the model is not fully covered and that event e_6 will never be mapped, selected, and executed by our tool. We call the event that is available in the GUI tree but does not exist in the model *an unknown event*.

To overcome this problem, it is necessary to adjust the probability distributions. We use *KenLM* [10], which is a fast and low-memory language model toolkit. It includes the modified Kneser-Ney [11] technique, which is considered to be an effective smoothing technique. The three key ideas of the technique are:

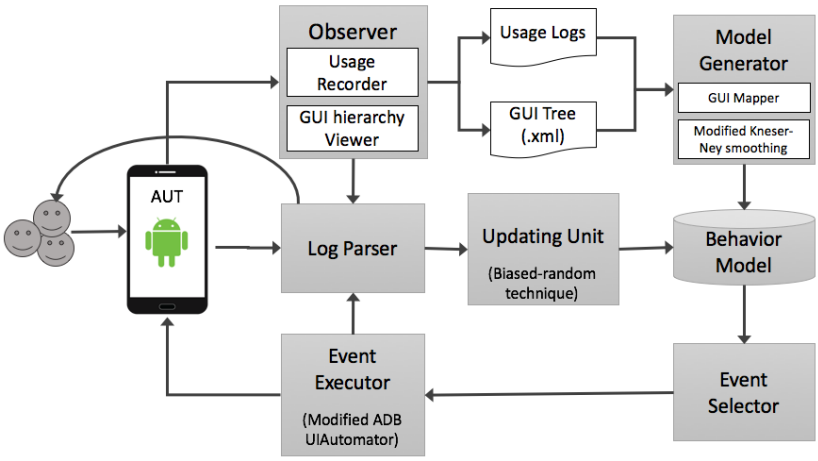

Fig. 1. An overview of the tool's architecture.

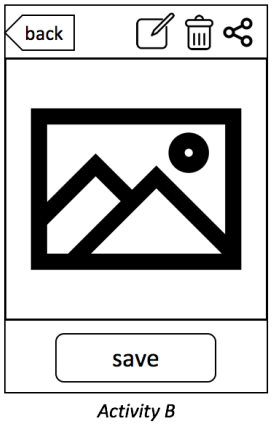

Fig. 2. The Example Application Under Test

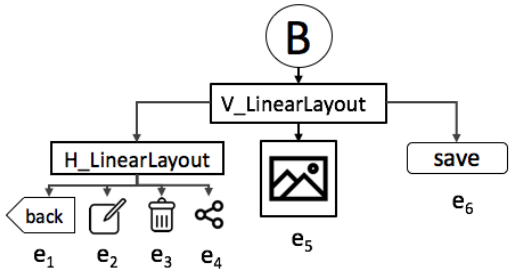

Fig. 3. A Simplified GUI Tree

- Absolute-discounting: in order to give an unknown event some probability, we must reduce the probability of the others in order to retain a valid probability distribution.
- Interpolation: recursively combine probabilities for all k-grams where $k \in 1, \ldots, n$. If an event sequence has any k-gram suffix appear in the model, it will yield a non-zero probability.
- Histories: the number of contexts that each event appears in should be taken into account. For instance, if an event occurs only in a specific context, it should be less likely to appear in a novel context.

Fig. 5 shows one example behavioral model after smoothing. The unknown event e_6, as well as rare events e_1 and e_5 receive some probability mass from the others.

2) Mapping behavioral model with GUI Tree: We do not actually merge an event and a GUI component together. Each event within the behavioral model is mapped to a GUI component within the GUI tree using a unique *resource-id* as a key, or with *type* and *text* if *resource-id* is not defined. The GUI mapper is implemented using a hash table for quicker lookup. The table is stored in main memory and updated each time after *Observation* (Section III-A). After selecting an event, the mapper looks up the table and returns the corresponding GUI fragment to the event.

C. Selection

The purpose of our tool is to test the application, not to create a natural action sequence. Therefore, the next event e_{next} will be randomly selected instead of naively picking the event with the highest probability. Let $e_1, e_2, e_3, \ldots, e_n \in \{possibleEvents\}$ and h be the history of previously selected events. e_{next} will be randomly selected by *Event Selector* based on the smoothed probability of an event given a sequence of previous selected events:

$$e_{next} = \omega([P(e_1|h), P(e_2|h), P(e_3|h), \ldots, P(e_n|h)])$$

where $\omega(L)$ is the weighted random function of a list L, $[P(e_1|h), P(e_2|h), P(e_3|h), \ldots, P(e_n|h)] \in L$, $\sum_{i=1}^{n} P(e_i|h) = 1$, and $P(e_x|h)$ is the probability of event e_x occurring given history h after updating (see Section III-E). From Fig 5, L should be *[0.07, 0.14, 0.28, 0.41, 0.07, 0.03]* for e_1 to e_6, respectively. We then calculate partial sum of the list to be *[0.07, 0.21, 0.49, 0.90, 0.97, 1.00]*. ω will uniformly random generate a number between 0.00 and 1.00 and select an e_x where the generated number is less than or equal to

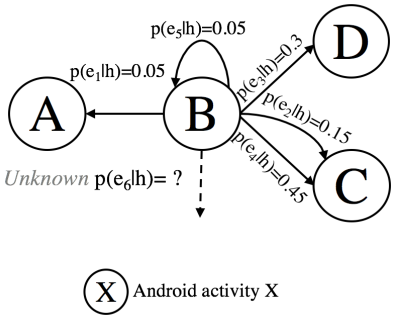

Fig. 4. A Simplified Behavioral Model

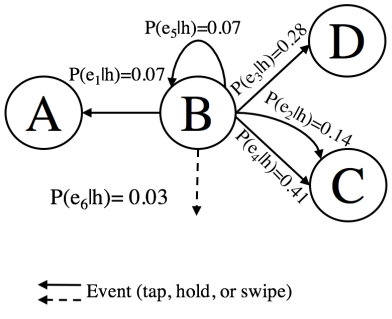

Fig. 5. Behavioral Model after smoothing

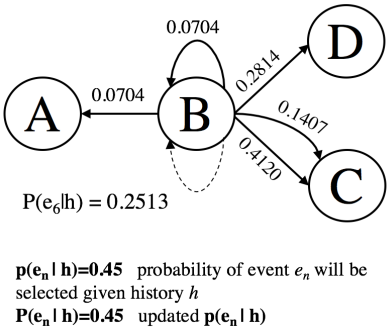

Fig. 6. Behavioral Model after updating

its partial sum but greater than e_{x-1}. Suppose the number is 0.99. In this case, e_{next} is e_6.

D. Execution

The *Event Executor* executes the e_{next} that was chosen by *Event Selector* on an actual mobile device or an emulator by *UIAutomator*, which is a custom wrapper around *Android Debug Bridge* written in python. Our tool can execute events such as Clicking, Long-Clicking, Scrolling, and Texting. In our tool, these events are defined as follows:

- Clicking: to tap and release immediately at the center of the view.
- Long-Clicking: to tap and hold 3 seconds before releasing at the center of the view.
- Scrolling: to tap and drag to $(x + \Delta x,\ y + \Delta y)$ position.
- Texting: to input an arbitrary pre-generated string[3]

Suppose e_{next} is the e_6 from Fig. 4, and it is clickable. The Event Executor will fire a *clicking event* to the center of the view, which in this case would be *saveButton*.

E. Updating

After execution, the model will be updated by using biased random technique to adjust selected/executed event's probabilities. It reduces a factor of small delta δ from the previously selected and executed event:

$$p\prime(e_x|h) = p(e_x|h) - d(e_x, \delta f_x) \tag{1}$$

where f_x is the frequency of e_x, $d(e_x, \delta f_x)$ is the discounted value of e_x, and $p\prime(e_x|h)$ is the probability of event e_x given history h. Suppose $d(e_6, \delta f_6) = 0.05$. Using equation (1), $p\prime(e_6|h)$ is 0.025.

After reducing some probability from an e_x, each $p\prime$ probability must be recalculated in order to retain a valid probability distribution by dividing it with total $p\prime$ probability.

$$P(e_x|h) = \frac{p\prime(e_x|h)}{\sum_{i=1}^{n} p\prime(e_i|h)} \tag{2}$$

The results of equation(2) are shown in Fig 6. The probability of $P(e_6|h)$ decreases. On the other hand, $P(e_1|h)$ to $P(e_5|h)$ slightly increases. This will allow other events to have a greater chance to be selected which should lead to higher code-coverage.

Finally, after executing e_6 and repeating the observation step (Section III-A), the *Updater* updates the behavioral model (Fig. 6).

[3]https://github.com/minimaxir/big-list-of-naughty-strings

TABLE II
LIST OF APPS USED IN OUR EVALUATION

App	Version	#LOC	Desc
Anymemo	8.3.1	8989	a flash card learning software.
World Clock	0.6	1251	a simple clock around the world app.
Weight Chart	1.0.4	1149	a log keeper application of body weight and display on a graphical chart.

IV. EVALUATION

We conducted a case study to evaluate our approach focusing on code coverage, and compared it to three popular approaches: Android Monkey, manual testing, and Dynodroid tool.

A. Target Application

We target three unmodified open-source Android applications: Anymemo[12], World Clock[13], and Weight Chart[14]. We chose these apps since they have dynamic content, static content, and complex user interface, respectively.

Table II shows the version numbers and short description of the applications.

B. Experiment setup

All our experiments were done using Android API 19, the current baseline version for application development, on Nexus 5's emulator with 1586 MB of RAM. The emulator was run on a 64-bit MacOS 10.12.4 machine with 2.5 GHz Intel Core i7 processors and 16 GB of RAM. The usage logs were collected by asking five computer science students to use each application for five minutes. No restrictions were placed on how they were to use the applications. For each experiment session, an AUT was installed on a freshly-created emulator

with only default setting. Each experiment session for our tool, Android Monkey, and Dynodroid consists of 5,000 touchable events and we added a delay between events of 500ms to ensure that screen transition or animation has completed before executing the next event. For manual testing, we asked an Android user to manually exercise these apps as much as possible within 40 minutes. After each experiment session, the emulator was destroyed to prevent it from affecting later sessions. We used Emma [15] and custom shell script to collect coverage measurement from AUT.

C. Results and Discussion

The results of our study are summarized in Table III. We collected code coverage at 500 event intervals for a total of 10 data collection points. The most left column denotes the testing approaches including our approach, Android Monkey, Manual testing by human, and Dynodroid, respectively. The numbers in the table show the number of lines and percentage of code coverage by total lines of code (see Table II) that were covered by each approach for the three applications tested. Our tool outperformed Android Monkey and Dynodroid in *Anymemo* but did not outperform manual testing.

Manual testing easily outperformed our tool and other approaches, achieving the highest coverage for all three applications. This can be expectable, given that the human can provide more intelligent text or sequential inputs. Fig. 7 shows the accumulated code coverage for *Anymemo*. The X axis denotes the number of events based on the 500 event intervals. The plots for manual testing are the final result since we did not record coverage during manual testing. This is because we did not want to disturb our human subjects during the session. Due to our behavioral model, our tool successfully completed a series of operations (in this specific case, reviewing flash cards), and was able to conduct further operations; but Android Monkey and Dynodroid could not attain this level of operations within 5000 events. However, our tool failed to generate a valid string for sensitive cases, for instance, import and export path, since our tool records only the event type, and not a particular string.

The results for *World Clock* and *Weight Chart* are just slightly higher than Android Monkey. Moreover, in *Weight Chart*, our tool significantly underperformed manual testing and slightly lower than Dynodroid. A possible issue might be the fact that our tool cannot detect and perform complex gesture such as pinching or dragging to the Canvas components [16]. For example, in *Weight Chart*, the possible actions for a *display graph*, a line graph for showing weight data, were not detected by our tool. This resulted in several actions to not be reached.

D. Threats to Validity

There are potential threats to validity of our results. The first threat to validity would be the use of students as the human subjects. Since the task was to compare the coverage from manual testing and other approaches, there is a possibility that professional testers are better at exercising AUT.

TABLE III
ACCUMULATED CODE COVERAGE AFTER 5000 EVENTS OR 40 MINUTES. THE STATS INCLUDE THE CODE COVERAGE FROM OUR TOOL (#COV), FROM ANDROID MONKEY (#MCOV), FROM MANUAL TESTING BY HUMAN (#HCOV), AND FROM DYNODROID (#DCOV)

App	Anymemo	World Clock	Weight Chart
#COV	2885 (32.1%)	1120 (89.5%)	608 (52.9%)
#MCOV	1856 (20.6%)	1023 (81.9%)	536 (46.6%)
#HCOV	4173 (46.4%)	1149 (91.8%)	986 (85.8%)
#DCOV	1327 (14.8%)	1096 (87.6%)	673 (58.6%)

The second threat to validity is the three applications we used. We chose the three because they were open-source, have several categories, and were used by various studies[5], [1]. However, the sizes of our subject applications are relatively small compared with the top applications in Google Play[4]. Further evaluation with larger applications should be conducted.

The third threat to validity is the usage logs for a behavioral model. We believe that testing the app casually vs seriously gives a different result. Since the benefit is not clear, we did not provide particular restrictions or goals during the recording sessions.

V. LIMITATIONS OF OUR TOOL

This section outlines the current limitations of our tool.

A. Minimum Android API level

Our tool works on Android API level 16(Version 4.1.x) and onward. However, this is not significant since only 2% of Android device are API level 15 and below[17].

B. Scrolled Events need to be throttled

Since the scrolled events are emitted from the Android's *AccessibilityService* constantly while the user is scrolling, a scroll event usually becomes repetitive events which disrupt the model's probability distribution. To prevent that from happening, our tool merges consecutive scroll events into a single scroll event. However, there is a chance that two consecutive but different scrolls are merged unintentionally.

C. Resource-id must be provided

It is possible that the GUI component's *resource-id* is not defined by developers. In this case, other attributes are used for mapping (see Section III-B2).

D. Canvas component's properties are not detected

We rely on *UIAutomator* to dump the GUI hierarchy during runtime. It fails to extract a canvas component's properties, which prevents our tool from executing corresponding GUI events.

[4]https://play.google.com

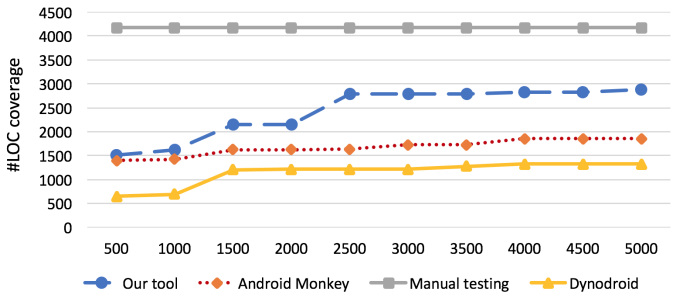

Fig. 7. Accumulated code coverage for Anymemo

VI. Conclusion and Future Work

In this paper, we presented our on-going work on automating GUI testing for Android applications. Our approach combines manual testing aspects by creating a behavioral model based on actual usage logs. We map the behavioral model with GUI tree in runtime, and handle the zero probability *unknown events* with *modified Kneser-Ney smoothing* technique.

We have implemented and conducted a comprehensive evaluation of our approach. We compared it with existing Android testing approach such as Android Monkey, manual testing, Dynodroid on open-source Android applications. Our approach outperforms Android monkey for all applications, and state-of-the-art tool Dynodroid for two out of three applications, but it is still behind manual testing.

For the future works, we will extend the actionable events that our tool can handle such as Enabling, Pinching, and Dragging. We also plan on extending our tool based on the limitations given in section V. We intend to evaluate our tool with larger Android application in the Google Play. Finally, we will further investigate our tool in term of time to reach the saturation point [18], and ability to find bugs.

Acknowledgement

This work was supported by JSPS KAKENHI JP15K00104.

References

[1] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet?" in *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 2015, pp. 429–440.

[2] M. E. Joorabchi, A. Mesbah, and P. Kruchten, "Real challenges in mobile app development," in *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, 2013, pp. 15–24.

[3] UI/Application Exerciser Monkey — Android Developers. [Online]. Available: http://developer.android.com/tools/help/monkey.html

[4] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using gui ripping for automated testing of android applications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2012, pp. 258–261.

[5] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: an input generation system for android apps," *9th Joint Meeting on Foundations of Software Engineering*, p. 224, 2013.

[6] P. A. Brooks and A. M. Memon, "Automated gui testing guided by usage profiles," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: ACM, 2007, pp. 333–342.

[7] J. L. San Miguel and S. Takada, "Gui and usage model-based test case generation for android applications with change analysis," in *Proceedings of the 1st International Workshop on Mobile Development*. ACM, 2016, pp. 43–44.

[8] 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 *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 111–122.

[9] M. Gómez, R. Rouvoy, B. Adams, and L. Seinturier, "Reproducing context-sensitive crashes of mobile apps using crowdsourced monitoring," in *Proceedings of the International Conference on Mobile Software Engineering and Systems*. ACM, 2016, pp. 88–99.

[10] K. Heafield. KenLM Language Model Toolkit. [Online]. Available: http://kheafield.com/code/kenlm/

[11] S. F. Chen and J. Goodman, "An empirical study of smoothing techniques for language modeling," *Computer Speech Language*, vol. 13, no. 4, pp. 359 – 394, 1999.

[12] H. Ning. AnyMemo. [Online]. Available: https://anymemo.org

[13] R. Agarwal. World Clock Android app. [Online]. Available: https://github.com/rahulaga/WorldClock

[14] Senselesssolutions. Weight Chart. [Online]. Available: https://github.com/bluezoot/weight-chart

[15] EMMA: a free Java code coverage tool. [Online]. Available: http://emma.sourceforge.net

[16] Canvas. [Online]. Available: https://developer.android.com/reference/android/graphics/Canvas.html

[17] Dashboards — Android Developers. [Online]. Available: https://developer.android.com/about/dashboards/index.html

[18] D. Amalfitano, N. Amatucci, A. R. Fasolino, P. Tramontana, E. Kowalczyk, and A. M. Memon, "Exploiting the saturation effect in automatic random testing of android applications," in *Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems*, ser. MOBILESoft '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 33–43.