Class Coverage GUI Testing for Android Applications

Sathyanarayanan Subramanian, Thomas Singleton, Omar El Ariss
Dept. of Computer Science and Mathematical Sciences
The Pennsylvania State University
Harrisburg, PA, USA
e-mail: {sps5564, tss5118, oelariss}@psu.edu

***Abstract*—Mobile devices such as smartphones and tablets have become an integral part of a person's life. These portable devices opened up a new software market for mobile application development resulting in various applications from healthcare, banking till entertainment. Therefore, there is a need for mobile applications to be reliable and maintainable. In this paper we introduce an equivalent class based technique for testing the graphical user interface of Android applications. This technique is a specification based approach, in which test cases are generated based on the functionalities and the graphical user interface specification. For each possible user interface event a set of test cases are generated using equivalence class partitioning approach. Once the test cases are generated for the given application, the app is executed based on the generated test cases and results are compared with the other testing techniques. From the obtained results we can infer that our approach detects more bugs than other previous work. In addition, this approach helps in the generation of test cases at an early in the app development life cycle.**

Keywords-Android; GUI testing; equivalence class partitioning; specification based approach

I. INTRODUCTION

Smartphones and Tablets have become one of the most important sources of access to information, where these hand-held devices for many users are now replacing most of the other computing devices [1]. These devices have rose to such significance through offering an abundance of applications in virtually any category one might think of. Hence these applications are capable of replacing other devices entirely, or featured enough to replace desktop applications for the average user. Modern mobile applications allow users to perform a variety of functions, from word processing, playing music, browsing the internet, to even turning a phone into a capable replacement for flashlights and standalone GPS devices.

Given the pervasiveness of mobile and tablet, it is increasingly important that testing strategies see rapid growth that brings mobile application testing in line with the much older and evolved software testing strategies. The nature of mobile applications requires testing methods to focus heavily on the reliability of applications [1]. This paper focuses on testing Android applications since Android is the most commonly used operating system in mobile devices, having approximately 61% of the mobile operating system market share as of April 2016 [2]. The Graphical User Interface (GUI) of Android applications works differently from a typical GUI-based desktop application: where a traditional desktop application handles interaction with the GUI via events, an Android application handles interaction with a user via its activities, Android services, and content providers which can share structured data between different applications [2], [3]. When an application is launched in Android, the Main activity is run by default and then other activities are called subsequently according to the user interaction. A created activity undergoes different states during its lifetime before it gets destroyed. Fig. 1 shows the dynamic behavior of a created activity. The main states for an Android activity are: *ActivityLaunched*, *Running*, *Paused*, *Stopped*, and *Destroyed*. But unlike regular applications, in Android OS there always will be only one activity in the visible state, while the other activities will be either stopped or in pause state. The 'Active' activity is the current activity that is displayed to the user. Once when the user navigates away from the current activity, then the activity ceases to be visible and moves to the Stopped state. If the current activity temporarily opens another view, say launching the camera as part of the current activity, then the current activity goes to the 'Paused' state.

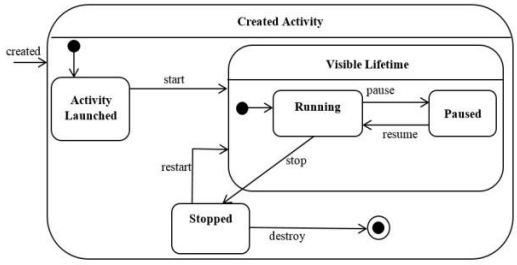

Figure 1. UML State machine diagram of a created activity in Android

As Maji et al. indicate in [4], the combination of issues results in a significantly higher average number of defects in mobile applications versus more traditional systems [4]. This necessitates the importance of testing strategies to focus on the GUI. Also testing the GUI should handle all the possible navigations a user can do like moving from one activity to another within the application, moving across multiple applications, rotating the same activity etc... Google provides several tools/testing frameworks to assist developers in testing their applications. Though these frameworks are an aid for testing, by applying testing strategies over these framework we can increase the quality of testing.

978-1-5090-3278-5/16/$31.00 ©2016 IEEE

Equivalence class partitioning is a software testing technique in which the input domain to the given software is divided into equivalence classes. Test case development proceeds by generating one test case per class until all the equivalence classes are covered [5]. Our proposed approach uses this testing strategy for testing mobile applications. The main advantage of this technique is that it covers all possible input values including valid and invalid input to widgets in a given activity. It also ensures that there are no redundant test cases covering the same input class. In addition, and more importantly, this strategy focuses on both valid and invalid input. This makes it possible to uncover bugs that are harder to discover by only focusing on the legal functionalities. The core idea of this paper is therefore to come up with a simple, easily maintainable, and efficient test coverage Android GUI testing strategy.

Considering the above objective, our approach is designed as follows. The application to be tested is given as an input, in apk format. Upon receiving the input, the proposed technique parses the apk file and identifies the .xml files that contain the necessary GUI details. Based on the information available in the .xml files, a list of activities in the given application are created and then the list of widgets and their corresponding input classes are identified for each activity. Once this list is generated, for each input class of a widget and for all the widgets in an activity, equivalence classes are identified and the test cases are designed. Then the test suite is executed and the logs are recorded during test execution. After test completion, along with the test results, the log files are analyzed and the information required for debugging is filtered out and given as output. Also the obtained results are compared with the Monkey tool, a command line tool for random testing an Android GUI, and test effectiveness is measured.

The rest of the paper proceeds as follows: Section II describes various mobile application testing techniques done so far. Then Section III explains the application of equivalence class testing to GUI testing. Section IV describes the model of the proposed testing system. Later Section V describes the results and Section VI concludes the paper covering future work and improvements.

II. Related Work

Mobile application testing is one of the active research area in Software Engineering. Mahmood, Mirzaei, and Malek [6] describe the implications of mobile application peculiarities to software testing, which includes mobile connectivity, limited resources, autonomy, Graphical User Interface (GUI), context awareness, and adaptation. This opens up various research areas in mobile application software testing. There are several research done on test automation for Android applications. The majority of them follow strategies to explore the application such as: model-based and specification based approach, systematic approach, random strategies, and capture & replay techniques [7]. Model-based approaches use tools such as web-crawlers [8]-[10] to build a GUI model of the applications under test. The model is then used to explore the application systematically in order to generate test cases [7]. Some of the tools that are based on this strategy are GUIRipper [11], MobiGUITAR [12], and A^{3}E-Depth-First [13].

Specification based approach on the other hand uses the system specification to come up with test cases [14]. System specifications are usually described in terms of specification languages, such as rule based approach and behavioral based approach. In rule based approach system specifications are given as a set of rules and these rules are applied as a classical planning techniques in artificial intelligent systems to come up with test cases for GUI applications [15]. In the behavioral based approach, specifications are given as system models, mostly as Finite State Machines (FSM). FSM represents the expected behavior of the given system, upon a user input or event [16], [17]. Our proposed approach follows this strategy.

The systematic approach is a technique in which the behavior of the application is explored by providing sample input to the application [7]. Android testing tools like A^{3}E-Targeted [13], EvoDroid [6] use this approach to explore the application. In some applications, certain events can be generated only by giving specific inputs, which necessitates the need for sophisticated techniques like symbolic execution and evolutionary algorithms [7]. EvoDroid uses evolutionary algorithms for generating inputs. Thus by implementing a systematic procedure one can reach the behavior of an application that is hard to reach through random approach. Nariman Mirzaei, et al. [18], proposed a tool called TRIMDroid, in which the tools used rules and constraints to generate test cases and control flow graphs for the generated test cases. Then, they reduced the number of test cases, by implementing Constraint resolver techniques.

Capture and Replay is an another technique which records the input events for the system under test and replays the recorded events later for testing purpose [19]. The main advantage using this approach is that mobile devices can get user inputs like swiping, touching, pinching, zooming, etc… which are difficult to generate using other approaches. Some of the tools like Reran [20], Mosaic [21], MobiPlay [19], uses this approach to generate test cases.

Random strategy on the other hand make use of random testing [22], which is the main engine of the approach, where input events to the application are randomly generated. This type of strategy is good for stress testing [7] and for generating a high volume of test cases. One major issue with the random strategy is that it becomes hard to generate specific type of input. Also testing methodologies based on this approach do not guarantee any test coverage, as it is purely random. Though some of the tools like Dynodroid [23] try to use biased Random strategy or a frequency strategy to explore new events, still the methodology is random in nature and cannot guarantee maximum coverage. Another GUI driven tool which comes along within the Android development toolkit is “Monkey” [24]. This is the most commonly used tool for testing an application as it is provided by Google. Monkey considers the application as a black-box and generates random UI events for a given duration of time.

Various research work has incorporated this tool in their approaches for better android application testing. In [1], Hu

and Neamtiu propose an efficient, debugging friendly, random approach for mobile application testing using the Monkey tool. This approach can be executed from command line by specifying whether to concentrate on the entire system or just a specific application, so that Monkey can generate events accordingly. If the Monkey is supplied a restricted number of applications, it will block attempts to access any other application. Monkey randomly performs most of the possible UI events like selecting the buttons in the activity, typing inputs to the activity, navigating to a different activity, volume controlling, etc. Additionally, Monkey will stop if it reaches any unhandled exceptions or causes the application to stop responding [24]. Hu and Neamtiu first start by manually identifying all the activities of an application. Then they use Monkey for each activity to automatically generate test cases for the System Under Test (SUT). These tests are then written using the JUnit framework [1]. Once the test cases are generated, Monkey is started in parallel along with the application log, which captures all the event in the system level. During testing, relevant details are gathered from the Android system log. Once the test process is completed, the log is given as an input to a log analyzer. This analyzer filters the log entries specific to the application and further gives the list of error messages thrown during test execution making it debugger friendly.

Apart from the above strategies, there is a need for an effective way to evaluate the proposed testing strategies. Fault injection is an approach in which faults are injected into the application's code and then the faulty application is tested. The goal of the generated test cases is to capture all the injected faults. If this is not the case, then new test cases are designed to fill those test gaps. Though this is one of the efficient way of analyzing test coverage, this method needs software code access, also it is difficult to model permanent faults [25].

We envision a GUI based testing strategy for mobile applications to be friendly, easy to implement, and open source since these applications are programmed by developers with various technical backgrounds. Our proposed strategy is a specification based approach that is loosely based on Hu and Neamtiu's work and make use of tools provided by the Android framework. In our approach, and rather than using Monkey and focusing on random testing, we introduce a specification based approach through the use of Espresso framework. Initially, for an application under test, we systematically define equivalence classes based on the possible events and the widgets in the activity/application. The equivalence classes are used to generate test cases that cover both valid and invalid input. The focus on invalid scenarios, or illegal behavior, makes this testing strategy a valuable one for GUI testing since it is possible to detect bugs that are hard to find. Once the test cases are defined, we use Espresso to execute the tests.

III. EQUIVALENCE CLASSES IN GUI TESTING

Equivalence class partitioning is a methodical way to effectively cover the input domain during the testing process without drastically increasing the size of the test suite. Although it is possible to apply equivalence partitioning to the output domain, this work only focuses on the input domain. This testing strategy divides the input of an Android application into several equivalence classes. Each equivalence class represents a subset of the input, where testing a value in this subset should be the same as testing any other value in the same subset. Next, we describe the steps needed to generate equivalence classes given an Android application.

In the Android operating system, an activity can have several widgets like buttons, text fields, checkboxes, radio buttons, toggle buttons, Spinners, pickers, menus, alert-dialogs, notifications, toasts, search, drag & drop, etc… Depending on the application, these widgets will have different valid and invalid equivalence classes. In this work we focus on the following widgets/events where we discuss the list of valid and invalid equivalence classes for each of them:

A. Radio Buttons

Valid Classes: one equivalence class for each radio button within a group. Only the radio button in the equivalence class should be selected by the test case. In case the user has already selected a different radio button, then the selected option should be unselected and the correct radio should be selected. *Invalid Classes*: (1) select multiple radio buttons within one group, (2) no radio buttons are selected.

B. Button

Valid Classes: There are four equivalence classes if the button is enabled: (1) Clicking and then releasing the button very quicly; (2) pressing and holding on the button for a few seconds then releasing it; (3) pressing the button normally; (4) pressing the button multiple times. There are no valid equivalence classess when the button is disabled. Invalid Class: click on the button when it is disabled.

C. Text Field

When given a minimum and maximum character limit then, *Valid Classes:* (1) enter the minimum limited number of characters; (2) enter the maximum number of characters; (3) enter the average number of characters within the limit; (4) enter only accepted (valid) character types, for example no special characters. *Invalid Classes:* (1) do not enter any characters; (2) enter one character less than the minimum character limit; (3) enter one character more than the maximum character limit; (4) enter both valid and invalid charracter with a valid size.

D. Spinner

Valid Classes: (1) scroll up the spinner then select an item; (2) scroll down the spinner then select an item; (3) select an item from the spinner then scroll up and down; (4) select more than one item from the spinner. *Invalid Classes:* (1) scroll down the spinner without selecting an item; (2) scroll up the spinner without selecting an item.

E. Menus

Valid Classes: An equivalence class for every menu item. The menu is first activated then the menu item is invoked

Invalid Classes: Two equivalence classes, one for the first menu item and one for the last item. A test case for each equivalence class will select the menu item but no menu events are clicked on, only tapped nearby over a non-menu.

F. System Events

While it is important to test the widgets of an activity, it is also important to test the behavior of an activity due to system operations like navigating to a different application, locking the device, rotating the device to portrait orientation (if the device is a landscape tablet), rotating to landscape orientation, leaving the system in idle state while the application is running, etc. One good example why to test these system states is to consider an application which uses map activity for navigation. *Valid Class* is when the user doesn't interact with the application during navigation, yet the application should be awake and should not time out and lock the device. *Invalid Class* will be, the application experiences idleness and locks the device, due to idle-time out. Another scenario to test the system operations is device rotation. Generally, mobile devices support the following rotation: 90 degrees, 180 degrees, 270 degree or 360 degrees and hence they are *Valid Classes*. *Invalid Class* includes, if a particular orientation is disabled in the activity and rotating the device to that orientation changes the display layout, then it is an invalid class.

In order to come up with test cases, one should identify all the equivalence partitions within the valid and invalid classes. For illustration, consider the previous example, in that among the valid equivalence classes there are Class 1: 90 degrees, 270-degree rotation which is a landscape orientation, Class 2: 180 degrees, 360-degree rotation is a portrait orientation. It is important to come up with a test case for each one of these equivalence classes because if we test any one value, say 90-degree rotation, and if the test passes, we cannot guarantee that even 180-degree rotation will pass, since the output is different for each of those rotation. This necessitates the need for test cases to cover all the equivalence classes among the identified valid and invalid scenarios.

IV. TESTING SYSTEM MODEL

Our proposed approach for Android GUI testing starts with the extraction of the activities and GUI widgets. Next we use equivalence class partitioning, which we discussed in Section III, to generate test cases from the GUI widgets. Once these test cases are designed, they are developed using JUnit framework and executed using Espresso framework. During the execution of the test cases, test logs are recorded in parallel and used for test report generation. All of these steps are done by three major modules: ***UI Extractor, Test Engine,*** and ***Log Analyzer***. Fig. 2, shows the high level design of our proposed architecture.

UI Extractor: Given an android application under test (AUT) as an input, this module extracts the list of activities, their UI components, and the necessary information for designing test cases from the .xml files. This extracted information is then passed as an input to the Test Engine. Fig. 3 shows the general structure of this module.

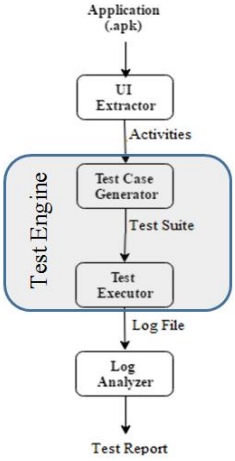

Figure 2. High level proposed architecutre for Android GUI testing

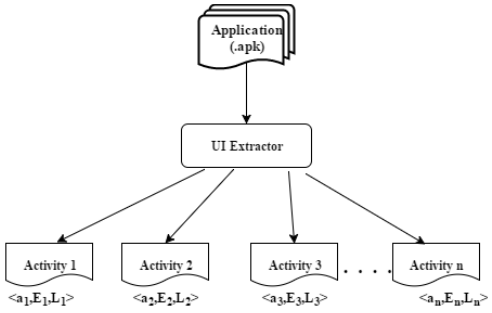

Figure 3. UI extractor module structure

Given the AUT as an input to the UI Extractor, the main step is to extract the list of activities *A* that contains the main activity (a_0) and zero or more other activities ($a_1, a_2, \ldots, a_n$). For all activities $a_i \in A$, where *n* is the total number of activities in A, each activity 'a_i' is represented as a tuple of three elements [18]:

$$\forall\ a_i \in A\ \exists\ <\boldsymbol{a_i, E_i, L_i}>$$

First element 'a_i' represents the activity id, in order to identify the activities and their associated widgets. Second, element 'E_i', represents the list of event handlers like onClick(), onPress(), etc… associated with the current activity. The final element 'L_i', is the list of widgets W_j and their associated input class 'I_k', where 'j' is the number of widgets in the current activity and 'k' is the number of input classes associated with each widget W_j. For example, if an activity has a text field and submit button, then number of widgets j = 2 and as we identified in section III, each button widget has k = 5 input classes, and the text field widget has k= 7 input classes (including valid and invalid ones).

Test Engine: The test engine step is the core module which efficiently executes the testing process. This module receives the activity information from the UI Extractor and generates test cases for each activity $a_i \in A$ and executes the test cases using Espresso framework. This process is achieved through the following two sub-modules:

1) Test Generator: This module reads the tuple information of all the extracted activities $a_i \in A$, and

generates the list of test cases, associated with each activity. For each identified activity 'a_i', there is a list of widgets 'W_j' where 'j' is the number of widgets in the given activity.

Each widget W_x, where x ∈ j, has a list of equivalence input classes 'I_k', where 'k' is the number of equivalence input classes for the given widget. Thus for each equivalence input class, a test case is generated. Once the test cases are designed for user inputs, the next step is to generate test cases for the system equivalence classes as described in Section III. The following procedure describes the above process:

i. *For each activity $a_i \in A$, where i ← 1...n and n is the number of activities in A*
   a. *Create $TC_i = \{\phi\}$, an empty set of test cases for the activity i*
   b. *For each Widget W_j in L_i, where j is the number of widgets in an activity*
      i. *For each Equivalence Input Class k ∈ I_j*
         1. *Create test case for the input class k as tc[k]*
         2. *tc[k] U TC_i*
   c. *Append System test cases to TC_i*
   d. *Append TC_i to the test suite TS*

ii. *return TS*

It is important to include system level test cases for each activity. Let us consider a music player app as an example. From the $<W_j, I_j>$ information, one can come up with test cases to play, pause, stop the music playback. But while the music is being played, if the user navigates to a different activity within the same application or to a different application, then the expected behavior will not be captured by the above test cases. In order to avoid lack of testing of such scenarios, it is necessary to include these test cases for all the activities $a_i \in A$. Thus once the test cases TC_i are designed for all the activities, they are passed to Test Executor for execution.

2) Test Executor: When the Test Executor receives the test cases TC_i, it models them as JUnit test cases in Espresso framework. The modeled test cases are then used for executing the application over the associated activity. While the test is being executed, Logcat logs are recorded in parallel. Logcat [26] is a command-line tool that is provided by android framework for logging system messages. After completing the entire test suite, for the given input application, recorded logs are sent to the Log Analyzer for test report generation.

Log Analyzer: This module analyzes the collected logs and identifies all the unique bugs, which are related to the AUT. In addition, it also reports the number of times a specific error is encountered during the test execution. This is achieved by using regular expressions to format the data dependent upon the AUT. This regular expression also filters out the unwanted details from the device's log, including Android system details and details pertaining to other applications running on the device.

V. RESULTS

Our proposed approach uses F-Droid, an Open Source app store for Android [27], for evaluating the test effectiveness. We have chosen six of the commonly used applications such as Flashlight, Photo-Manager, Droid-Gain, Another Monitor, Call Recorder, Calendar-widget. Then we tested these applications using our approach, along with the commonly used android Random testing tool, Monkey. We also compared our results along with the number of known bugs reported by the application developers in the F-Droid store. Results clearly shown a better improvement in terms of test effectiveness.

For illustration consider the Flashlight application. One use case of the application is that, upon a button click, the device's flashlight turns on and again by clicking the same button, the flashlight turns off. Our approach identifies the classes as follows. This app has only one activity, $a_1 \in A$. The number of event handlers in a_1 is 1 that is onClick(), and the number of Widgets is 1, a single button. But this widget has the following input classes: When the flash light is off, the flash should turn-on upon a button press; a long press on the button should also turn on the flash. When the flash light is On, the flash should turn-off upon another button press; a long press on the button should also turn off the flash. As a result, we identify two equivalence classes: Class 1 and Class 2. Along with the above identified classes, system input classes 3, 4, and 5 are also appended to generate one Test case for each classes. The identified equivalence input classes are:

- Class 1-Turn on flash
- Class 2-Turn off flash
- Class 3-Navigate to different application
- Class 4-Kill application
- Class 5-Device rotation

Once the test cases are executed on this application using our proposed approach, we then tested the application using the Monkey tool. Monkey didn't identify any new bugs apart from the three already known bugs. On the other hand, our approach, identified two new bugs. One is that, when the flashlight is turned on and the device is rotated, the flashlight flickers during rotation. Another bug is that, while navigating to a different app when the flash is on, the expected behavior is that the flash should remain on, until it is turned off. But immediately upon navigation, the flash turns off. The expected behavior is compared and confirmed with the system flash app.

Similar to the Flashlight example, we have identified equivalence classes for each application. Based on the identified classes, GUI test cases are then generated for each application under test. Fig. 4 shows our testing results compared to the use of Monkey. With the use of our proposed approach, the generated test cases identified either the same number of bugs or in most cases more bugs. Looking at the bar-chart of Fig. 4, it is evident that our proposed approach generates test cases that handles unexpected scenarios or boundary conditions that lead to the discovery of additional bugs that were not identified by the app developers or by the Monkey tool.

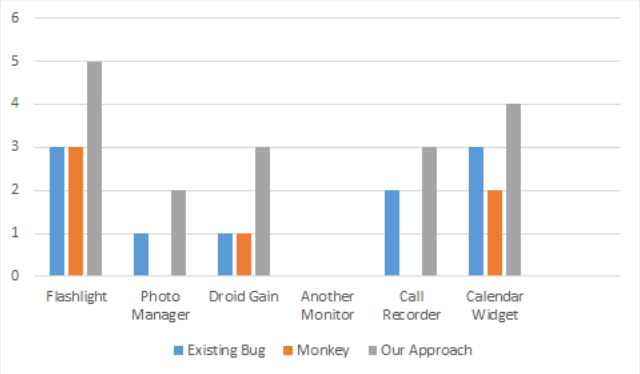

Figure 4. Graphical representation of bugs identified by our approach compared with Monkey and previously identified bugs by app developers

VI. Conclusion & Future Work

Due to the rapidly expanding tablet and smartphone market, it is increasingly important that new and improved testing strategies are developed for Android devices. Testing of mobile and tablet applications require techniques that are more suitable to the unique aspects associated with these devices. We proposed a GUI based testing strategy based on the use of equivalence class partitioning. Since our proposed approach uses equivalence class coverage technique, one can come up with GUI test cases immediately after designing the specification. Hence it can be used in the early stages of the software life-cycle. Also our approach is systematic in exploring the test-cases, which is easily adaptable when the application is modified, and therefore qualifies as a robust testing technique. Along with these advantages, our approach helps with the maintenance of the app, as it filters necessary error information related to testing the application. In the future work, we will focus on automating the UI Extraction module, as we currently explored the app manually. In addition, more invalid boundary value analysis will be investigated to capture corner cases. Finally, we will look at the benefits of generating equivalence classes from the output domain.

References

[1] C. Hu and I. Neamtiu, "Automating gui testing for android applications," *6th Int. Work. Autom. Softw. Test (AST 2011)*, no. Section 4, pp. 77–83, 2011.

[2] "Operating system market share." [Online]. Available: https://www.netmarketshare.com/operating-system-market-share.aspx?qprid=8&qpcustomd=1.

[3] "Getting Started | Android Developers." [Online]. Available: https://developer.android.com/training/index.html.

[4] A. K. Maji, K. Hao, S. Sultana, and S. Bagchi, "Characterizing failures in mobile OSes: A case study with android and symbian," *Proc. - Int. Symp. Softw. Reliab. Eng. ISSRE*, pp. 249–258, 2010.

[5] Aidas Kasperavičius IFM-0/2, "Equivalence Partitioning."

[6] R. Mahmood, N. Mirzaei, and S. Malek, "EvoDroid: segmented evolutionary testing of Android apps," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*, 2014, pp. 599–609.

[7] S. R. Choudhary, A. Gorla, and A. Orso, "Automated Test Input Generation for Android: Are We There Yet?," *Ase*, pp. 429–440, 2015.

[8] V. Dallmeier, M. Burger, T. Orth, and A. Zeller, "WebMate: Generating test cases for web 2.0," *Lect. Notes Bus. Inf. Process.*, vol. 133 LNBIP, pp. 55–69, 2013.

[9] S. R. Choudhary, M. R. Prasad, and A. Orso, "X-PERT: Accurate identification of cross-browser issues in web applications," *Proc. - Int. Conf. Softw. Eng.*, pp. 702–711, 2013.

[10] A. Mesbah, A. van Deursen, and S. Lenselink, "Crawling Ajax-Based Web Applications through Dynamic Analysis of User Interface State Changes," *ACM Trans. Web*, vol. 6, no. 1, pp. 1–30, 2012.

[11] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using GUI Ripping for Automated Testing of Android Applications," *Ase*, pp. 258–261, 2012.

[12] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "MobiGUITAR: Automated Model-Based Testing of Mobile Apps," *IEEE Softw.*, vol. 32, no. 5, pp. 53–59, Sep. 2015.

[13] T. Azim and I. Neamtiu, "Targeted and Depth-first Exploration for Systematic Testing of Android Apps," *Oopsla*, pp. 641–660, 2013.

[14] J. Chen and S. Subramaniam, "Specification-based Testing for GUI-based Applications," *Softw. Qual. J.*, vol. 10, no. 3, pp. 205–224, 2002.

[15] A. Memon, I. Banerjee, and A. Nagarajan, "GUI ripping: reverse engineering of graphical user interfaces for testing," in *10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings.*, pp. 260–269.

[16] L. White and H. Almezen, "Generating test cases for GUI responsibilities using complete interaction sequences," *Softw. Reliab. Eng. 2000. ISSRE 2000. Proceedings. 11th Int. Symp.*, pp. 110–121, 2000.

[17] R. K. Shehady and D. P. Siewiorek, "A method to automate user interface testing using variable finite state machines," in *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*, pp. 80–88.

[18] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek, "Reducing combinatorics in GUI testing of android applications," *Proc. 38th Int. Conf. Softw. Eng. - ICSE '16*, pp. 559–570, 2016.

[19] Z. Qin, Y. Tang, E. Novak, and Q. Li, "MobiPlay," in *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*, 2016, pp. 571–582.

[20] L. Gomez and T. Millstein, "RERAN : Timing- and Touch-Sensitive Record and Replay for Android," pp. 72–81, 2013.

[21] M. Halpern, Y. Zhu, R. Peri, and V. J. Reddi, "Mosaic: cross-platform user-interaction record and replay for the fragmented android ecosystem," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2015, pp. 215–224.

[22] A. F. Tappenden and J. Miller, "Random Testing," *IEEE Trans. Reliab.*, vol. 58, no. 4, pp. 619–633, 2009.

[23] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: an input generation system for Android apps," *Proc. 2013 9th Jt. Meet. Found. Softw. Eng. - ESEC/FSE 2013*, p. 224, 2013.

[24] "UI/Application Exerciser Monkey | Android Studio." [Online]. Available: https://developer.android.com/studio/test/monkey.html

[25] H. Ziade, R. Ayoubi, and R. Velazco, "A Survey on Fault Injection Techniques," *Int. Arab J. Inf. Technol.*, vol. 1, no. 2, pp. 171–186, 2004.

[26] "logcat Command-line Tool | Android Studio." [Online]. Available: https://developer.android.com/studio/command-line/logcat.html

[27] "F-Droid – Free and Open Source Android App Repository." [Online]. Available: https://f-droid.org/