日本ソフトウェア科学会第 35 回大会 (2018 年度) 講演論文集

Applying Combinatorial Strategies in GUI Testing Incorporating Context Events in Android Applications

Maria Azriel Therese Eala　Shingo Takada

Android applications interact with users mainly through widgets that trigger GUI events. In addition, smartphones have other components, such as sensors and location services, that may greatly affect the behavior of the application. The input generated by these additional components are referred to as context events. This paper proposes an approach for test case generation that takes into account both GUI and context events. However, incorporating both GUI events and context events into test case generation can lead to combinatorial explosion. This approach uses combinatorial strategies to generate the sequences that make up the test cases to address the combinatorial explosion problem given a significant number of inputs. This approach greatly reduced the number of test cases compared to an exhaustive combination approach while still getting high code coverage compared to state-of-the-art tools.

1 Introduction

The number of mobile applications is significantly high. Recent statistics show there are around 5.5 million mobile applications available for both Android and Apple mobile devices. Aside from this, mobile applications present new challenges in software testing as it is quite different from traditional applications such as web. Hence, research on this field has increased.

In Android mobile application testing, most approaches tend to focus on the Graphical User Interface (GUI) events. Wasserman [15] and Muccini et al. [14] cited some specifications in mobile applications that are not applicable to traditional software. These specifications include events produced by additional components that provide contextual or sensed data, and user interface guidelines and specifications. These should be taken into consideration when testing mobile applications.

However, combinatorial explosion can result when generating test cases that cover all interactions between the inputs, e.g., both GUI events and contextual events. Combinatorial testing approaches have proved to be effective for GUI testing. We propose an approach for testing Android applications using combinatorial methods to generate test cases including GUI events with sensor events.

The remainder of this thesis is structured as follows: Section 2 gives a brief background of Android mobile applications and development, and combinatorial interaction testing, Section 3 discusses related works on Android testing, Section 4 explains the methodology behind the proposed approach.

2 Background

2.1 Android

Android is an operating system for mobile devices which offers an Application Programming Interface (API) for developing applications. Android mobile applications mostly interact with the users via a User Interface (UI) screen called an `Activity`. Since this is the main gateway for an application, most approaches tend to focus on these inputs alone.

However, an Android mobile application can use other components and features available on an Android device and the API is used to leverage these

エヤラ　マリヤ　アズリエル　テリス、高田眞吾, 慶應義塾大学理工学研究科, Graduate School of Science and Technology, Keio University.

features. Features such as sensors, connectivity, and location services can be used by an Android mobile application. Although the API may be sufficient enough to create applications, it is limited when it comes to testing contextual events. For example, the API provides methods that can allow an application to listen and react to sensor events, however, it does not allow for mocking sensor values that may be necessary when testing.

2.2 Combinatorial Interaction Testing

According to Cohen et al. [9], faults may be caused by an unexpected interaction between certain parameters' components, therefore, all possible interactions should be tested. However, as a software becomes more complex, they tend to have more parameters and components. This results in a large number of test combinations that lead to combinatorial explosion. And running all possible combinatorial test cases is not possible due to limited time and resources. Combinatorial interaction testing (CIT) has been used for finding faults in software systems by generating only a subset of all possible interactions of components.

The idea behind CIT is based on orthogonal arrays or covering arrays. The basic concept behind the arrays is that a certain interaction of components should be covered, or included, in exactly the same number of test cases (for orthogonal arrays) and in at least one test case (for covering arrays). These arrays adhere to t-way coverage criteria which means that all possible combinations of t parameters should be included in the test suite. However, as t approaches the number of parameters, ending in all-value combinations or the exhaustive approach, more time and resources are needed to generate the test suite.

According to Kuhn et al. [12], however, failures are caused by interactions of a few components. Therefore, tests that cover these interactions may be effective. The results of their study showed that all failures were triggered by a maximum of 4-way to 6-way interactions, $t = 4$ to 6.

Kuhn et al. [11] also introduced another type of array called sequence covering array which is more apt for creating event sequences. It eliminates the parameters and instead creates a covering array of sequences of non-repeating components. This approach is also based on t-way coverage.

3 Related Work

Android Monkey [10] is a random-based GUI testing tool. It is a command line tool that is packaged with the Android developer toolkit for testing Android applications. It sends a pseudo-random stream of GUI events and a number of system-level events to the Application Under Test (AUT). It is easy to use and readily available which is why it is the most widely used tool to test Android applications.

AndroidRipper [6] [8] is a GUI crawling tool used for testing Android applications by dynamically exploring the application to extract GUI events and infers a GUI model during breadth-first traversal. The tool follows three steps: (1) GUI ripping, (2) test case generation, and (3) test case execution. The model is created during the GUI ripping step.

The two tools mentioned above focuses on generating test cases with GUI events only. These do not consider additional inputs, i.e. contextual events, which could also be handled by an application.

ExtendedRipper [7] is an extension of the *AndroidRipper* tool that takes into consideration contextual events, aside from just GUI events. Like *AndroidRipper*, *ExtendedRipper* dynamically explores the application to detect user events. Based on analysis of bug reports from open-source applications, they manually identified event-patterns to be triggered during test execution. The tool detects explicit contextual events from the application, which are events that are handled by the application. These events are then mixed with the GUI events either randomly or systematically.

Yu et al.'s approach [17] [18] takes into consideration, not only explicit contextual events, but also implicit contextual events in testing Android applications. Explicit events are identified by static analysis of the application and implicit events are determined by keywords specified by the tester as well as events deemed to be related to the identified explicit events. However, the test cases are executed manually.

With *ExtendedRipper*, although it considers contextual events, it only considers explicit event detected by the tool. These exclude other contextual events, undefined in the application, that may trig-

ger faults and affect the application. With Yu et al's approach, the test cases are manually executed which would take a significant amount of time if there is a large number of test cases.

TrimDroid [13] is a tool that generates test cases using GUI events based on combinatorics. The tool statically analyzes the source code to define the dependencies for generating test cases. These dependencies determine the valid and possible combinations of events that should be included in the test suite. A sequence of events is generated by taking into account the GUI event and input values for some events.

GUI interaction testing proposed by Yuan et al. [19] is an approach that generates test cases that follow t-way coverage between events. The proposed approach leverages covering arrays, similar to sequence covering arrays discussed in Section 2, to develop a new technique in constructing GUI test cases. The approach uses an event-interaction graph (EIG) to model the relationship between GUI events and is used to obtain the covering arrays. The events are then manually partitioned according to functionality, and constraints are identified to make sure the sequence of events are valid.

These two approaches uses combinatorial techniques to generate test cases, but only takes into consideration GUI events.

4 Proposed Approach

We propose an approach that generates and executes test cases for Android applications. The approach handles events generated by other components in an Android mobile device, such as sensor events, and not just conventional GUI events. The approach also makes use of combinatorial methods to address the combinatorial explosion problem that may arise with the increase in the size of the input events.

There are five components to this approach: GUI modeling, GUI event sequence generation, context event sequence generation, test case generation, and test case execution. The following is the summary of the proposed approach as seen in Figure 1:

1. **Generate Context Event Sequences** –Context event sequences are generated given the defined context events in the repository.
2. **Extract UI Information** –The Android application's source code is explored to generate a GUI model for GUI event sequence generation.
3. **Generate GUI Event Sequences** –GUI event sequences are generated using the information provided in the GUI model.
4. **Generate Test Cases** –GUI event sequences and context event sequences are combined to create valid test cases.
5. **Execute Test Cases** –Test cases are automatically translated into Android code and then executed on an Android device.

4.1 GUI Model

We used GATOR [16] [3], an open-source toolkit that statically analyzes the souce code of the application to extract the transition flow between windows by identifying events and callbacks in the application. Figure 2 shows an example of a model extracted from an application. The model extracted using GATOR serves as the constraints for generating valid GUI event sequences.

Static analysis of the application's source code is done to extract GUI widget map, containing necessary GUI widget information, which is used for the sequence generation and the translation of the test cases into Android code.

4.2 Event Sequence Generation

One of the key ideas of the proposed approach is to use combinatorial strategies to generate event sequences and the final test cases to reduce the size of the test suite and avoid combinatorial explosion which can occur given a large number of events. This section describes the sequence generation for both GUI events and context events.

4.2.1 GUI Event Sequence Generation

We take into consideration the transition graph, extracted using GATOR, and the GUI widget map to create GUI event sequences.

If the exhaustive approach was applied to generate event sequences, it would generate n^l combinations, where n is the number of events and l is the length of the sequence. This number could get excessively large if the number of events or the length of the sequence increased. This is the combinatorial explosion problem our proposed approach wishes to address.

Our approach creates a sequence covering array

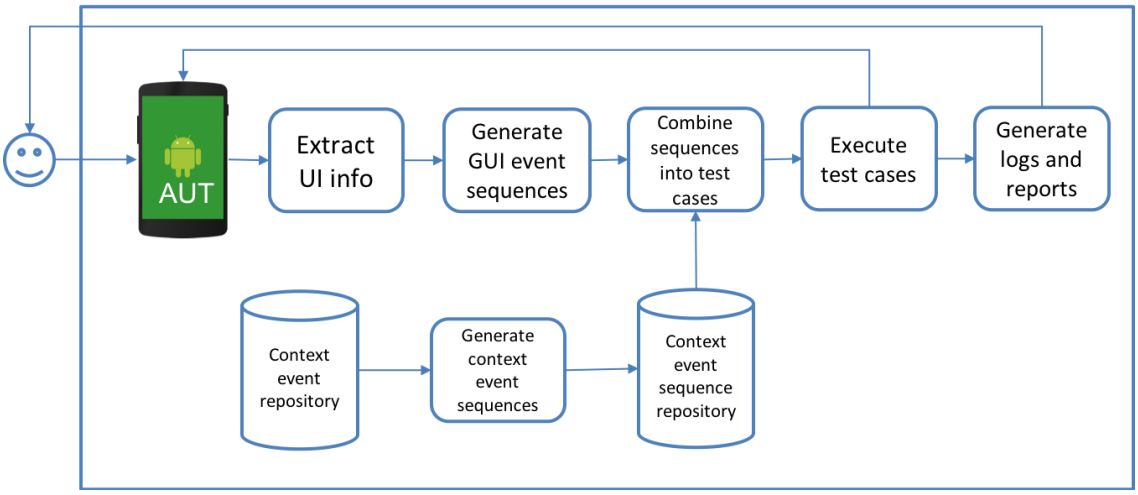

Fig. 1 Proposed Architecture

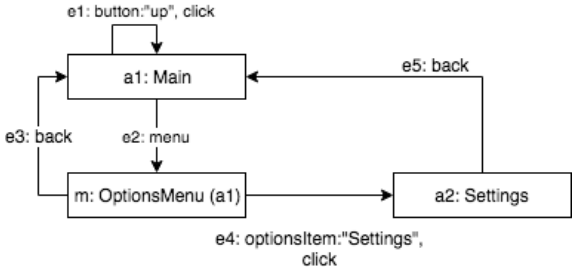

Fig. 2 Example of a GUI Model

to generate GUI event sequences, where the events may be repeated in a sequence. The t-way coverage is used for this approach, where t is l-1 and l is the length of the sequence. The proposed approach generates sequence covering arrays from length three to length ten (i.e. l=3–10).

4.2.2 Context Event Sequence Generation

Another key idea of the proposed approach is to incorporate context events in testing Android applications. Aside from GUI events, Android applications take in as inputs, events generated by additional components present in the device, e.g. sensor events and location services. These additional events should be also be considered when testing Android applications.

A repository of context events and their restrictions have been manually identified by Yu et al. [17] [18]. This study leverages the available information from the Context Event repository created by Yu et al. However, only sensor and location events are considered in this study.

The same steps used to generate GUI event sequences were used to generate context event sequences. When generating context event sequences, our approach does not consider repeating events, i.e. one event is executed only once in a sequence, and the maximum length of the sequence is six. Context event sequences are generated per context event category, hence, should the number of context events in a category be less than six, the maximum length of the sequence would be the total number of events. Interleaving of events in context event generation may also be considered as part of future work.

Context event sequences are generated only once and are then stored in a database. Context event sequences are only re-generated if and when new context event categories and context events are identified. This reduces the time to generate test cases since the context event sequences are generated beforehand and are only retrieved from the database when injecting it into the GUI event sequences.

4.3 Test Case Generation

We apply combinatorial methods when creating the actual test cases, which are combinations of the GUI event sequences and context event sequences.

We do not combine the GUI event sequences directly. As each application activity can only handle

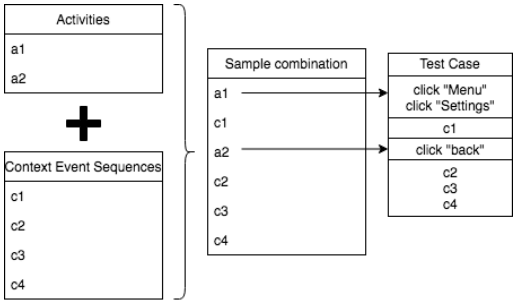

Fig. 3 Combining Activities and Context Event Categories

a certain subset of GUI events, we first combine activities and context event sequences.

Figure 3 shows an example of combining activities and context event sequences. In this figure, we consider an application with two activities and four context event sequences.

An example combination of the activities and context event sequences would be *<a1, c1, a2, c2, c3, c4>*, where *c1*, *c2*, *c3*, and *c4* are context event sequences and *a1* and *a2* are activities. The GUI events in a generated GUI event sequence belong to specific activities. For example, in Figure 2, activity *a1* has the GUI event sequence *<click "Menu", click "Settings">*, while activity *a2* has *<click "back">*. The context event sequences are then injected just before the transition between activities. Thus, one generated test case is *<click "Menu", click "Settings", c1, click "back", c2, c3, c4>* which is shown in Figure 3.

The number of context event sequences included in the test cases was 4. This is an arbitrary value and can be changed.

4.4 Test Case Execution

4.4.1 GUI Events

This section discusses the translation of the test cases into Android code. The Android development kit includes Espresso, a programming interface to create Android UI tests [2]. The Espresso framework provides methods to perform valid GUI events on the application. The approach uses the Espresso framework to execute GUI events on the application.

For every GUI event in the test case, there are corresponding methods available in the Espresso API. These methods take in either the widget's ID or text to determine which GUI widget to perform the action on.

4.4.2 Context Events

The Android framework does not have methods that would allow testers to send mock sensor values. However, there is an existing open-source library that modified the `android.hardware` package to mimic sensor events. OpenIntents SensorSimulator [5] is an open-source software which provides a library for Android applications to mock sensor values. The software transmits simulated sensor data to an Android emulator from a server.

There is one big difference between defining the use of the SensorSimulator and the built-in `android.hardware.Sensor*` package, and that is the instantiation of the Sensor Manager.

This links all the succeeding definition of listeners to the *SensorSimulator* library instead of the `android.hardware.Sensor*` package.

To implement the *SensorSimulator*, however, a server is needed to receive simulated sensor values. Connecting to a server is unnecessary as sensor values can be generated without the use of an external software. Therefore, the *SensorSimulator* library was modified to remove and replace the methods and lines of code relating to server connection and communication. These were replaced with code that can generate mock sensor values to be handled by the application.

The application under test is then modified to adapt to the *SensorSimulator* library. The application under test was modified to call the `SensorManagerSimulator` from the *SensorSimulator* library instead of the `SensorManager` from the `android.hardware.Sensor*` package. The instantiation for the `SensorManagerSimulator` is described above.

For location events, the Google Play Service Location API allows mocking a device's location. However, only mocking of location is possible. Setting the availability of the Location Provider is not allowed.

5 Evaluation

5.1 Test Procedure

5.1.1 Evaluation Setup

The Android application Pedometer [4] is a free and open-source application available on GitHub

and was used to evaluate the proposed approach. This application is a simple pedometer that tracks and records the user's steps. We used the latest version, where the latest commit was on 2016/03/15.

20 GUI events were identified during the static analysis of the application. 21 context event categories were included.

The application was installed and tested on a physical Android device with Sony Xperia X2 Dual running on Android Nougat, v.7.1.1. The tool is run on MacPro with 3.7 GHz Quad-Core Intel Xeon E5 with 32GB RAM.

The code coverage was generated and parsed using JaCoCO [1], a code coverage tool for Java programs. JaCoCo is a gradle plug-in that is used by the Android Studio IDE.

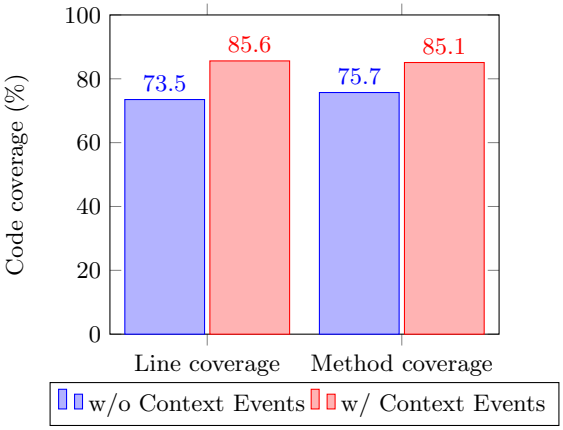

Fig. 4 Code Coverage Comparison in Context Event Inclusion

5. 2 Results and Discussion

This study aims to answer the following research questions:

- RQ1 How significant is the reduction in generation time and evaluation time?
- RQ2 How does including context events in testing compare to the GUI events-only approach?
- RQ3 How do the code coverage of the proposed approach and the coverage of state-of-the-art tools differ?
- RQ4 How effective is the proposed approach in detecting faults?

The results of the evaluation and discussion of the research questions are addressed in this section.

5. 2. 1 RQ1: Test Case Reduction

The approach aims to reduce the number of test cases compared to the exhaustive approach. This is concerned with reducing the number of generated GUI test cases, not including context events in the test cases. Table 1, where `n` is the length of the sequence and s is the number of sequences generated, compares the number of GUI event sequences generated by using the exhaustive approach and the proposed approach. The GUI event sequences were generated for the Pedometer application with 20 GUI events.

The approach was set to generate GUI event sequences until `l` = 10. However, the computer used for evaluation threw a `java.lang.OutOfMemoryError: Java heap space` error due to the large amount of combinations being generated. Due to this limitation, the approach only generated sequences with `l` = 6 for the exhaustive approach and `l` = 7 for the proposed approach.

Reducing the number of test cases would also mean reducing the text execution time. On average, using the development machine, it takes around two hours per 1000 test cases. For l = 6, 584,000 test cases for the exhaustive approach would take roughly 49 days to execute, while it would only take 5 days for the less than 60,000 test cases generated by the proposed approach.

The approach only generated an average of 11.1% of the number of GUI test cases compared to the exhaustive approach.

5. 2. 2 RQ2: Context Event Inclusion

21 context event categories were used by the proposed approach, 20 of which were sensor events. The sensor event categories only include one event per category. As for the location events, only sending of mock location is permitted by the Android and Google Play Store Service APIs. Therefore, our approach only handled one event for the location category.

With the inclusion of context events in the GUI test cases, the approach achieved around 12% increase in both line coverage and method coverage as shown in Figure 4. The Pedometer application handles events from the Accelerometer sensor which was triggered by the approach, hence the increase in code coverage.

The missed 15% in code coverage may be due to other GUI event information that were not ex-

Table 1 Number of Generated GUI event sequences for Pedometer

	Exhaustive approach		Proposed approach	
l	s	time (in mins)	s	time (in mins)
3	560	0.0012	68	0.0012
4	5, 444	0.0245	616	0.0014
5	55, 432	1.4161	5, 920	0.0348
6	584, 032	348.039	59, 744	3.6026
7	N/A	N/A	625, 472	470.6577

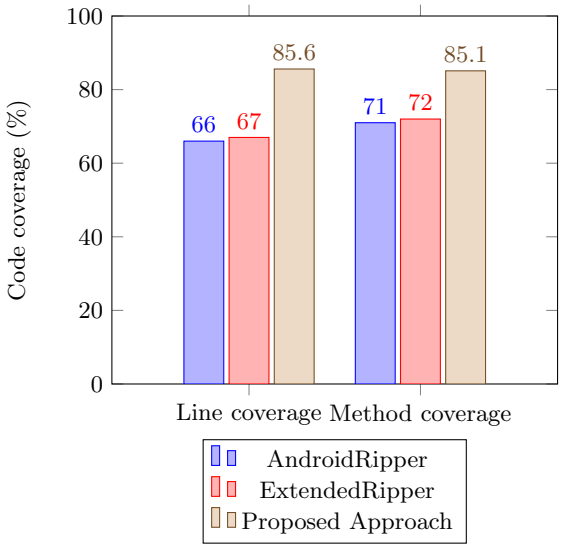

Fig. 5 Code Coverage Comparison between tools

tracted during static analysis. Static analysis was done to extract the GUI widget map, however, our implementation currently cannot extract widgets such as custom widgets. Also, there was one class in the application source code (`StepBuzzer.java`) which is only used for debugging purposes and was not traversed during test execution.

5. 2. 3 RQ3: Code Coverage

The coverage in this approach is compared to the coverage generated by both the *AndroidRipper* tool and the *ExtendedRipper* The results were taken from Amalfitano et al.'s [7] paper based on the *ExtendedRipper*.

The reduced test suite generated by the proposed approach managed to gain higher code coverage than tools like the *AndroidRipper* and *ExtendedRipper*. The proposed approach gained 29.7% and 27.76% increase in line coverage and a 19.86% and 18.19% increase in method coverage from the *AndroidRipper* and *ExtendedRipper*, respectively. Figure 5 shows the code coverage percentage for each tool.

5. 2. 4 RQ4: Fault Detection

There were no faults detected during the testing of the Pedometer application with the proposed approach.

A second experiment was done to validate the fault detection ability of the proposed approach. A version of the Pedometer application was seeded with ten faults. Ten conditional mutation operators were injected into one faulty version of the application. Eight of the ten faults were found using the proposed approach.

The second experiment proved that the proposed approach was able to detect faults. However, how well it detects faults is not guaranteed.

5. 3 Threats to Validity

The main threat to the validity of this evaluation is the number of applications considered for testing. Only one application was used for the evaluation of the tool, however, for future work, more applications are being considered to be used for evaluation.

6 Conclusion and Future Work

We proposed an approach to generating test cases for mobile applications, specifically Android applications. The approach handles conventional GUI events as well as events triggered by additional components in an Android device called context events. The context events considered in this approach are sensor events and location service events. Context event sequences are generated from these context events and are merged with generated GUI event sequences.

Future work includes the following:

- Include more context events for testing.

- Interleave context events from different categories in generating context event sequences.
- Consider repeating events and lengthier context event sequences.
- Interleave individual context events with GUI events in generating test cases to mimic natural behavior.
- Create a library, similar to *OpenIntents SensorSimulator*, for location services, etc. to simulate more context events.
- Include more applications for the evaluation of the proposed approach.

7 Acknowledgment

This work was supported by JSPS KAKENHI JP15K00104.

References

[1] EclEmma - JaCoCo Java Code Coverage Library, `https://www.eclemma.org/jacoco/`.

[2] Espresso | Android Developers, `https://developer.android.com/training/testing/espresso/`.

[3] GATOR: Program Analysis Toolkit For Android, `http://web.cse.ohio-state.edu/presto/software/gator/`.

[4] Github - bagilevi/android-pedometer, `https://github.com/bagilevi/android-pedometer`.

[5] SensorSimulator, `https://github.com/openintents/sensorsimulator`.

[6] Amalfitano, D., Fasolino, A. R., and Tramontana, P.: A GUI Crawling-based Technique for Android Mobile Application Testing, *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, 2011, pp. 252–261.

[7] Amalfitano, D., Fasolino, A. R., Tramontana, P., and Amatucci, N.: Considering Context Events in Event-Based Testing of Mobile Applications, *Proceedings of the 2013 IEEE International Conference on Software Testing, Verification and Validation Workshops*, 2013, pp. 126–133.

[8] Amalfitano, D., Fasolino, A. R., Tramontana, P., De Carmine, S., and Memon, A. M.: Using GUI Ripping for Automated Testing of Android Applications, *Proceedings of the 2012 IEEE/ACM International Conference on Automated Software Engineering*, 2012, pp. 258–261.

[9] Cohen, M., Gibbons, P., Mugridge, W., and Colbourn, C.: Constructing Test Suites for Interaction Testing, *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, 2015, pp. 429–440.

[10] Developers, A.UI/Application Exerciser Monkey, `https://developer.android.com/studio/test/monkey`.

[11] Kuhn, D. R., Higdon, J. M., Lawrence, J. F., Kacker, R. N., and Lei, Y.: Combinatorial Methods for Event Sequence Testing, *Proceedings of the 5th International Conference on Software Testing, Verification and Validation*, 2012, pp. 601–609.

[12] Kuhn, D. R., Kacker, R., and Lei, Y.: SP 800-142. Practical Combinatorial Testing, Technical report, Gaithersburg, MD, United States, 2010.

[13] Mirzaei, N., Garcia, J., Bagheri, H., Sadeghi, A., and Malek, S.: Reducing Combinatorics in GUI Testing of Android Applications, *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 559–570.

[14] Muccini, H., Di Francesco, A., and Esposito, P.: Software Testing of Mobile Applications: Challenges and Future Research Directions, *Proceedings of the 7th International Workshop on Automation of Software Test*, 2012, pp. 29–35.

[15] Wasserman, A. I.: Software Engineering Issues for Mobile Application Development, *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, 2010, pp. 397–400.

[16] Yang, S., Zhang, H., Wu, H., Wang, Y., Yan, D., and Rountev, A.: Static Window Transition Graphs for Android, *Proceedings of the 2015 IEEE/ACM International Conference on Automated Software Engineering*, 2015, pp. 658–668.

[17] Yu, S. and Takada, S.: External Event-Based Test Cases for Mobile Application, *Proceedings of the 2015 International C* Conference on Computer Science & Software Engineering*, 2015, pp. 148–149.

[18] Yu, S. and Takada, S.: Mobile Application Testing Focusing on External Events, *Proceedings of the 1st International Workshop on Mobile Development*, 2016, pp. 41–42.

[19] Yuan, X., Cohen, M. B., and Memon, A. M.: Covering Array Sampling of Input Event Sequences for Automated GUI Testing, *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, 2007, pp. 405–408.