Data Driven Testing for Context Aware Apps

Ryan Michaels
St Edward's University
rmichael@stedwards.edu

Shraddha Piparia
University of North Texas
ShraddhaPiparia@my.unt.edu

David Adamo
Block, Inc.
dadamo@squareup.com

Renee Bryce
University of North Texas
Renee.Bryce@unt.edu

***Abstract*—Context driven environments are growing in popularity. Mobile applications, Internet of Things devices, autonomous vehicles, and future technologies respond to context events in their environments. This work uses a set of context events from real users to guide the generation of context driven test cases. Context event sequences are obtained by applying Conditional Random Fields (CRF). Test suites are then constructed by interleaving the context event sequences with GUI events. The choice of context event is made based on transitions obtained from the CRF. Results of the empirical studies show that techniques that incorporate context events provide better code coverage than NoContext for the subject applications. A heuristic technique introduced in this work, ISFreqOne, yields 4x better coverage than NoContext, 0.06x better coverage than Random Start Context, 0.05x better coverage than Iterative Start Context, which are control context generation techniques, and 0.04x better coverage than ISFreqTwo, another heuristic introduced in this work.**

***Index Terms*—Android Testing, Context events, GUI events, Software Testing, Test Suite Generation**

I. Introduction

Many applications allow streams of context and user events to influence their behavior. We see examples in the domains of autonomous vehicles, Internet of Things (IoT), and mobile devices. For instance, if a user clicks a button, an app that responds by accessing data over the Internet to update the user's view will respond differently the device's context changes from WiFi to airplane mode. Additional examples of context events include changes in battery levels, the device's physical location, sound output to speakers or headphones, and changes to screen orientation. The work in this paper generates context aware test suites with a strategy based on a real-world data set of context events [1].

In the remainder of the paper: Section II summarizes related work; Section III describes the event sequence model; Section IV covers the data driven test generation strategy; Section V describes experiments; Section VI shows results and discussion; Section VII shares threats to validity; and Section VIII gives conclusions.

II. Related Work

GUI testing: GUI testing is an important task that many tools support. Examples include Monkey [2], Dynodroid [3], and Autodroid [4]. Most tools generate test cases without consideration of context events and their interactions with the application under test (AUT). Tests are often generated with respect to an initial setup of context variables that do not change during the testing. This may result in insufficient exploration of application states and code.

Monkey [2] offers fast and replayable test cases in the form of random clicks and swipes. Monkey does not interact explicitly with on-screen elements such as buttons and text fields, but clicks on events at specific screen coordinates [2].

Dynodroid [3] is an online testing tool that is responsive to application changes when it generates a "next event". It considers both system and user generated events. In an extensive study, Dynodrod generated 50 open source applications and outperformed Monkey in terms of code coverage.

Tema [5], [6] is an online GUI testing framework that utilizes models of application behavior informed by user data to generate abstract tests and are independent of device platforms. The models identify application state from abstract user actions. In its final step, Tema translates the abstract test cases into test cases by mapping of actions for specific devices and applications.

Context-aware GUI testing: Testing that uses both context and GUI events may increase fault detection for context driven apps [3], [7]–[10]. Dynodroid [3] is one tool that considers context events. While generating context events, Dynodroid does not offer a guarantee of combinatorial coverage of context and GUI event intersections will occur.

Adamsen et al. [11], Majchrzak et al. [12], and Song et al. [10] each propose context-sensitive mobile testing approaches. The approaches execute test suites under differing context environments. The change in context can change a valid test into an invalid one, such as a video streaming app attempting to execute without WiFi. While their approaches cover multiple context environments, there are opportunities to improve with more cost-effective strategies.

Amalfitano et al. [7] use context and GUI event combinations into test case generation. They consider a small set of GUI and context events and interleave them during test generation. The work demonstrates benefits of testing with both context and GUI events.

Griebe et al. [8] use a model-based approach where testers generate an annoted UML diagram to describe GUI behavior and context parameters of the AUT. The authors later expand this approach to incorporate sensor input [13]. Using the UI sensing tool Calabash-Android, they generate sensor values into the test cases [14]. They further provide parsing of natural language expressions, such as "I invert the phone" to generate additional test data.

DOI reference number: 10.18293/SEKE2022-125

CAIIPA [15] is a cloud based testing service that supports context aware apps. Their results demonstrate that using real-world context events during test generation improves performance fault and crash detection up to 11x better than Monkey. CAIIPA utilizes real-world context data. However, it is limited to hardware options such as WiFi and sensor settings. It cannot detect context such as screen orientation. AppDoctor [16] is another a cloud-based automation testing tool that injects events such as changes to network states, device storage, GUI gestures, and more during execution.

MoTiF [17] identifies and replicates context sensitive crashes for Android apps. Future work may extend this to not only reproduce crashes but to fully identify information of crash patterns across applications.

MobiCoMonkey [18] extends Monkey [2] by itnerleaving context events at random or as predefined by testers. This tool could be enhanced in the future with systematic interleaving of context and GUI events.

Conditional Random Fields (CRFs): Hidden Markov Models(HMMs) [19] are popular probabilistic sequence models [20]. However, HMMs suffer difficulty in modeling arbitrary, dependent features of input sequences. To overcome this drawback, we apply conditional random fields (CRF) [21] to testing context-aware systems. CRF sequence models are discriminative in nature, which allows for maximization of conditional likelihood.

III. Event Sequence Model

Our event sequence model is built on Autodroid [22] which comes with a test builder, abstraction manager to identify GUI events available in the AUT at different states, an event selector, and the event executor. We add context events to enable dynamic context-GUI testing for context aware applications.

Let us define GUI and context actions and events.

Definition 1. We define individual context events as a 2-tuple (c, a); c is a context variable with a as the assigned context action.

Screen orientation	WiFi	Battery	AC power
portrait	connected	Ok	connected
landscape	disconnected	low	disconnected
-	-	high	-

TABLE I
EXAMPLE CONTEXT VARIABLES AND POSSIBLE VALUES

Table I shows four context variables (screen orientation, WiFi, battery status, AC power) that may be set to any of the respective values shown in the table. A context event is then defined as one of these variables with an assigned value, i.e. c= ScreenOrientation = *landscape*, WiFi = c*disconnected*, Battery = *low*, Power = *disconnected*.

Definition 2. Action: We define an *action* as a user interaction with the application *or* a system level call with a target. Each action consists of a *target*, *type*, and *value*. There are two types of actions:

- **GUI action:** A user executes an action using GUI widgets, e.g. a click on a on-screen widget or filling in a text-field.
- **Context action:** The mobile operating system executes a system action, e.g. change in screen orientation.

Sometimes an action may require a *value*. For example, a text box may require users to type in a value. We consider two (or more) actions to be equivalent if they have the same target and type.

Definition 3. Event: We define an *Event* consists of a sequence of actions with pre and post conditions. A GUI event causes a change in the GUI state after its execution. A context event has one or more context actions and may or may not cause a change in the GUI state. An event is a complete event if it is executed and its post condition is known, otherwise it is a partial event.

Definition 4. Test Case: We define a test case as a sequence of events. Each test case has a unique id, a set of events, and a length. Table II shows a test case of length two containing one context event and one GUI event.

ID	tc001
Event 1 (GUI event - launch)	**event**: **precondition**: **activity name**: null **state id**: null **postcondition**: **activity name**: MainActivity **state id**: 07f24 **actions**: **type**: launch **value**: null **target**: **selector**: system **selector value**: app **type**: app **description**: launch
Event 2 (Context event - Power disconnected)	**event**: **precondition**: **activity name**: MainActivity **state id**: 07f24 **postcondition**: **activity name**: MainActivity **stateId**: 5d158 **actions**: **type**: power_disconnected **value**: null **target**: **selector**: power **selector value**: disconnected **type**: context **description**: power disconnected

TABLE II
A TEST CASE WITH ONE CONTEXT EVENT AND ONE GUI EVENT

IV. Data Driven Test Generation Strategy

Data collection and CRF modeling. To generate the CRF model, we observed everyday smartphone use from 58 university students for the duration of one month, via a monitoring app [1]. The app was configured to listen to 144 broadcasts which occurred 16,257,795 times during the one month period. [1]. This data fed into the construction of CRFs using

Fig. 1. Pseudocode for data-driven test generation algorithm based on Autodroid framework [23]

Inputs: Android app. package (*AUT*), combinatorial context model (*M*), context event sequence (*C*) context event frequency (*p*)

Outputs: Test suite (T)

1: $C_{all} \leftarrow$ *generateContextsFromCoveringArray*(M)
2: $T \leftarrow \phi$ ▷ test suite
3: **repeat**
4: $T_i \leftarrow \phi$ ▷ test case
5: $e_{context} \leftarrow$ *selectInitialContext*(C_{all})
6: $T_i \leftarrow T_i \cup e_{context}$
7: install AUT and execute launch event, e_{launch}
8: $T_i \leftarrow T_i \cup e_{launch}$
9: $s_{curr} \leftarrow$ initial GUI state after app launch
10: **while** *termination condition is false* **do**
11: $E_{all} \leftarrow$ *getGuiEvents*(s_{curr})
12: $e_{sel} \leftarrow$ *selectGuiOrContextEvent*(s_{curr}, E_{all}, C, p)
13: execute selected event, e_{sel}
14: $T_i \leftarrow T_i \cup e_{sel}$
15: $s_{curr} \leftarrow$ current GUI state
16: **end while**
17: *tearDownTestCase()*
18: $T \leftarrow T \cup \{T_i\}$
19: **until** *completion condition is false*

142,138 instances for training data and 28,663 instances of context events for test data.

The CRF is the graph obtained after considering the top likely transitions. We only consider the dependency between a predefined subset of events and remove all other external dependencies. Our aim is to find a context sequence which represents these dependencies without self-loops. We then select the transitions with the highest weight.

Test generation algorithm. The algorithm in Figure 1 uses context event sequences derived from a CRF to generate a sequence of context and GUI events, and then saves the generated sequence as a test case. The test generation algorithm consists of the following inputs:

- A compiled Android application
- Context event sequence, *C*, obtained from the CRF
- An integer value, *p*, that determines how often a context event will be added to a test case

The following discussion walks through the algorithm:

Step 1: Generate context covering array. Line 1 uses the combinatorial context model provided as input to generate a covering array. Each entry in the covering array represents a possible starting context for one or more test cases.

Step 2: Test case setup. Lines 4-9 represent the test setup for each test case. Line 4 initializes the test case as an empty event sequence. At the beginning of each test case, line 5 chooses an initial context from the covering array generated in step 1 and applies the chosen context to the execution environment. Line 6 adds the starting context event to the test case. Line 7 installs the AUT and launches it in the starting context. Line 8 adds the launch event to the test case. Line 9 retrieves the initial GUI state of the AUT after launch.

Step 3: Select and execute an event. Line 11 identifies all GUI events that are available and executable in the current GUI state. Lines 12-14 use the current GUI state, the context event sequence C derived from the CRF, and the specified context event frequency p to choose which GUI event or context event to execute.

The algorithm repeats steps 1 to 3 until a specified test case termination condition holds true. Context events are added to the test case at the predefined frequency p until the test case ends or until all events in the context event sequence, C, have been executed.

Step 4: Test case teardown. After the algorithm executes the last event in each test case, line 17 resets the test environment to allow tests to run independent of each other. The algorithm repeats steps 1 to 4 to generate multiple test cases containing a mix of context events and GUI events until a specified test suite completion condition holds true.

V. Experimental Setup

For our experiments we consider four applications, described in Table III. The applications have between 1,215 - 15,062 lines of code, 197-1134 methods, and 46-209 classes. Each application has over 1,000 downloads at the time of our experiment. The application's apk files are downloaded from F-Droid [24].

App Name	Installs	Vers.	Lines	Methods	Classes
Diode	10k+	1.3.2.2	7,933	1134	209
Your Local Weather	5k+	5.6.4	15,062	499	114
MovieDB	1k+	2.1.1	2,719	319	81
Abcore	1k+	0.77	1,215	197	46

TABLE III
Characteristics of the apps under test

A. Experimental Setup

The experiments use the Android 10.0 Pixel emulator (API 29) to generate ten test suites of two hour duration for each technique and application in the study. A two second delay occurs between event executions and there is a .05 probability to terminate a test case. We instrument subject applications with JaCoCo [25] to measure coverage.

B. Variables and Measures

Independent Variable. The independent variable of our experiments is our test generation technique. We consider three control techniques and two heuristic techniques.

Control techniques:

- **NoContext** generates a test suite of GUI events with only an initial set of context variables that do not change during testing. *NoContext* construct test suites

using c = *ScreenOrientation=Portrait WiFi=connected, Battery=OK, AC Power=connected*$\}$.

- **RandomStartContext (RSContext)** starts each test case by selecting a start context at random from a context covering array and then makes random selections of only GUI events.
- **IterativeStartContext (ISContext)** starts each test case by selecting a start context in a round-robin fashion from a context covering array and then makes a random selection of only GUI events.

Heuristic techniques:

- **IterativeStartFreqOne (ISFreqOne)** iterates through the context covering array to set a starting context and then uses context sequences obtained from the CRF to interleave context events with GUI events at an *interval of one* until all events in the context sequence have been executed.
- **IterativeStartFreqTwo (ISFreqTwo)** iterates through the context covering array to set a starting context and then uses context sequences obtained from the CRF to interleave context events with GUI events at an *interval of two* until all events in the context sequence have been executed.

The heuristic techniques, *ISFreqOne* and *ISFreqTwo*, use context event sequences derived from a CRF that includes only events for internet connectivity, power connection, battery level, and screen orientation changes.

Dependent Variables. We assess our research questions using code coverage:

- **Line coverage** measures the number of lines of code executed by the test suite relative to the total number of statements in the AUT.
- **Method coverage** counts the number of methods executed by the test suite relative to the total number of methods in the AUT.
- **Class coverage** counts the number of classes executed by the test suite relative to the total number of classes in the AUT.

C. Research Questions

The experiments examine two research questions:

RQ1: Do ISFreqOne and ISFreqTwo increase line, method, and class coverage in comparison to NoContext, RSContext, and ISContext?

RQ2: Which of the two heuristic techniques provide the greatest coverage of lines, methods, and classes in the test applications?

VI. Results and Analysis

RQ1 Results: Tables IV, V, and VI show the average results for line, method, and class coverage for each technique and app. We calculate the ratio of our techniques to the controls by dividing average values of both heuristics (ISFreqOne, ISFreqTwo) by control techniques (NoContext, RSContext, and ISContext). The ISFreqOne technique shows an improvement of approximately four times (312%) line coverage, three times (191%) method coverage, and 3.5 times (251%) class coverage when compared to NoContext for the application AbCore. IsFreqOne shows an improvement of 1.2 times (17%) line coverage, 1.2 times (21%) method coverage, and 1.2 times (18%) class coverage for ISFreqOne when compared to NoContext for the application MovieDB. ISFreqOne shows about 4% increase in line, method, and class coverage when compared to NoContext for the application Diode. ISFreqOne does not result in improvements for Your Local Weather relative to NoContext. For the Your Local Weather application, method coverage values for ISFreqOne and NoContext are similar although the class coverage is less for ISFreqOne when compared to NoContext. Your Local Weather has low overall code coverage. Our tool could not explore the application much because it needs a location to be entered or selected from the map which hindered exploration. The average improvement of ISFreqTwo over NoContext is 1.8 times line coverage, 1.5 times method coverage, and 1.67 times class coverage across all four applications.

	NoContext	**RSContext**	**ISContext**	**ISFreqOne**	**ISFreqTwo**
Abcore	15.83	62.87	63.95	65.15	62.83
MovieDB	40.65	45.22	45.45	47.5	47.205
YourLocalWeather	9.05	9.09	9.04	9.05	9.03
Diode	32.44	32.89	33.33	33.69	33.34

TABLE IV
AVERAGE LINE COVERAGE

	NoContext	**RSContext**	**ISContext**	**ISFreqOne**	**ISFreqTwo**
Abcore	25.38	71.72	72.88	73.81	71.28
MovieDB	47.37	54.12	54.55	57.365	55.175
YourLocalWeather	15.32	15.33	15.15	15.18	15.15
Diode	43.81	44.08	45.37	45.50	44.32

TABLE V
AVERAGE METHOD COVERAGE

On average, ISFreqOne offers improvement of 3.03% line coverage, 3.35% method coverage in comparison to RSContext across all four applications for ISFreqOne over RSContext. There is an average improvement of 2.2% line coverage, 0.6% method coverage, and 1.1% class coverage for ISFreqTwo over NoContext. Across all the applications, there is 2.2% line coverage, 2.14 method coverage, and 0.17% class coverage improvement in ISFreqOne over ISContext. We observe an improvement of 1.1% line coverage and 0.92% class coverage in ISFreqTwo over ISContext across all four applications. The method coverage, on average, did not show improvement for ISFreqTwo over ISContext. The noted improvements in code coverage indicate that for context-aware mobile applications, the presence or absence of context events in test suites has noticeable effects on the behavior of applications and the ability of test suites to adequately explore application functionality. The results also suggest that context events in test suites are useful not just at the beginning of test cases but also mixed in with GUI events at different intervals within each test case.

RQ2 Results: We compare results obtained from both heuristic techniques: ISFreqOne and ISFreqTwo. ISFreqOne outperforms ISFreqTwo across two subject applications. For Movie DB, there is an improvement of 0.62% line coverage

	NoContext	RSContext	ISContext	ISFreqOne	ISFreqTwo
Abcore	21.74	76.30	77.33	76.24	76.39
MovieDB	51.36	59.88	59.45	60.49	61.11
YourLocalWeather	27.93	27.93	25.88	25.88	25.88
Diode	38.42	39.27	40.08	40.10	39.98

TABLE VI
AVERAGE CLASS COVERAGE

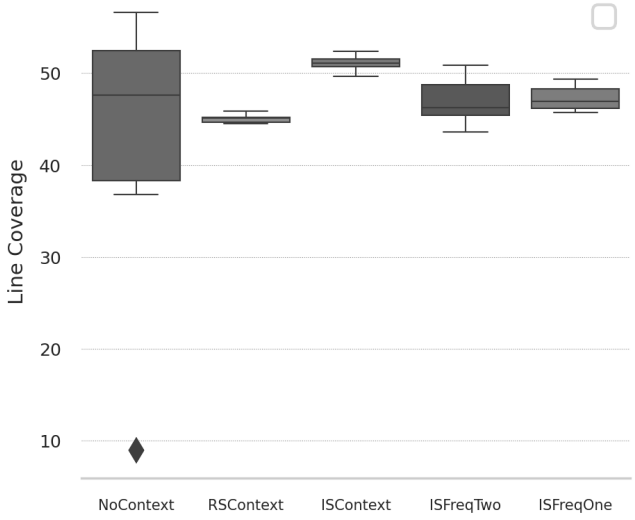

Fig. 2. Application *Movie DB*: Box plot of NoContext, RSContext, ISContext, ISFreqTwo, and ISFreqOne

and 3.9% method coverage. Class coverage is 1.03% higher for ISFreqTwo compared to ISFreqOne. The application AbCore shows an improvement of 3.7% line coverage and 3.5% method coverage for ISFreqOne over ISFreqTwo. Class coverage is 2.1% higher for ISFreqTwo over ISFreqOne. Diode application shows an improvement of 1.18% line coverage, 3.17% method coverage, and 0.42% class coverage for ISFreqOne over ISFreqTwo. The application Your Local Weather performs similarly for both of these techniques. On an average, there is an improvement of 1.15% line coverage and 3.76% method coverage for ISFreqOne over ISFreqTwo with 0.18% less class coverage. These results indicate that inserting context variables at an interval of one GUI events shows more coverage than inserting at two GUI events. This could be due to the fact that the Android applications are small in size. Figures 2 - 5 show the box plot the line coverage of all five techniques for Movie DB, AbCore, Diode, and Your Local Weather respectively. The standard deviation is less for ISFreqOne when compared to ISFreqTwo for all applications, indicating ISFreqOne is the more reliable generation heuristic on the AUTs in this study.

VII. Threats to Validity

The subject applications used for the experiments have different characteristics which may affect the performance of our techniques. Adding more subject applications can help to better generalize the results. We tried to minimize this threat by choosing applications of varying sizes from

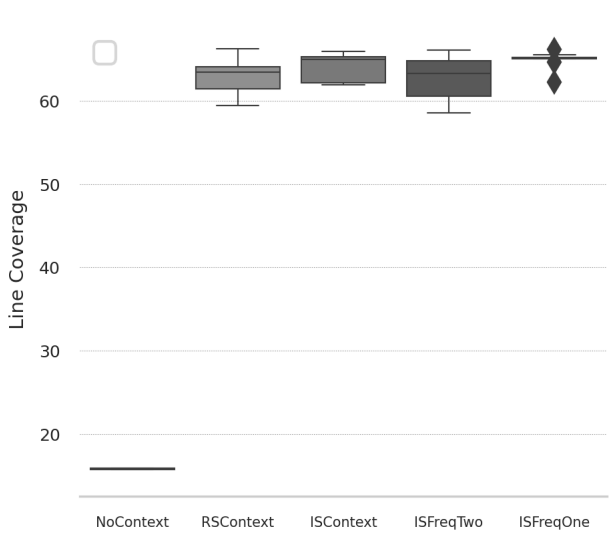

Fig. 3. Application *AbCore*: Box plot of NoContext, RSContext, ISContext, ISFreqTwo, and ISFreqOne

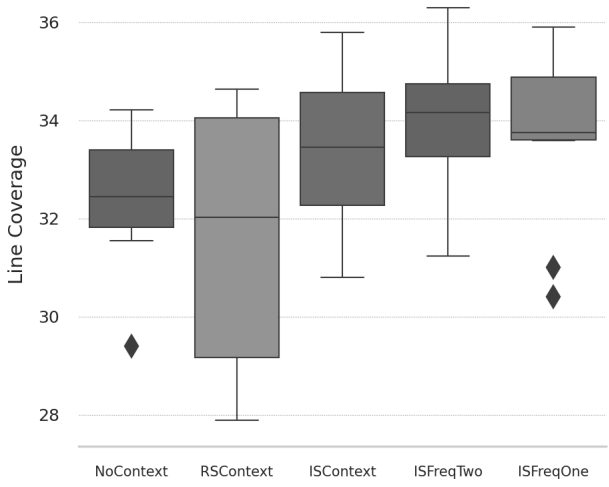

Fig. 4. Application *Diode*: Box plot of NoContext, RSContext, ISContext, ISFreqTwo, and ISFreqOne

different domains. The set of context events used in this experiment are limited in number and do not cover the totality of context events in Android. We minimized this threat by focusing on events that are reliably accessible in the emulator. We specifically excluded sensor events due to the sheer amount of data produced by sensors and the battery-intensive operations required to collect a constant stream of sensor values which helps in minimizing this threat. Inclusion of more context events may change results. For instance, we did not consider internet changes in this work. This was because the values in our data contain several internet connections such as HSPA_CONNECTION, HSDPA_CONNECTION, LTE_CONNECTION, etc. We hope that future advancements in emulators or inexpensive device

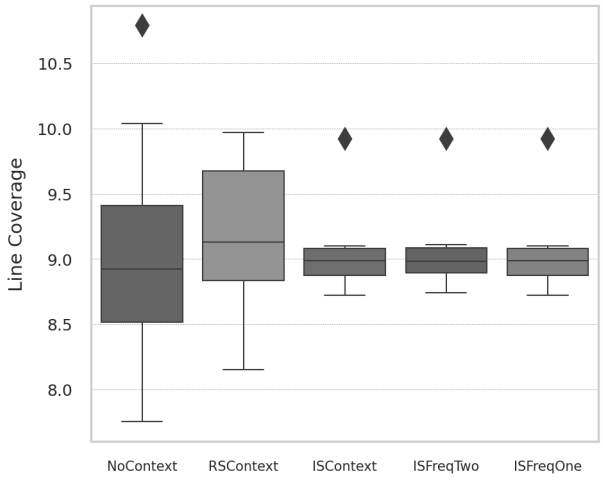

Fig. 5. Application *Your Local Weather*: Box plot of NoContext, RSContext, ISContext, ISFreqTwo, and ISFreqOne

farms will allow such expansions.

VIII. Conclusions

This work develops and studies algorithms that systematically generate test suites comprised of context events and GUI events. In particular, these techniques are guided by a real world data set of context data from 58 users. The choice of context events is made based on transitions obtained from CRF. We analyze how often context change should occur by providing an interval as a parameter to the tool. We observe that the techniques that incorporate context events performed better than NoContext among all four subject applications. The heuristic technique ISFreqOne yields four times better coverage than NoContext, 0.06 times better coverage than RSContext, 0.05 times better coverage than ISContext, and 0.04 times better than ISFreqTwo strategies. ISFreqOne also has a lower standard deviation between runs. The initial context strategies RSContext and ISContext performed better than NoContext by up to a factor of four indicating the importance of context while testing Android applications.

Future work will extend this study to examine larger context data sets and apply the techniques to more mobile applications. This work also serves as a foundation for future work that extends our techniques to other domains such as Internet of Things (IoT), wearable technologies, and autonomous vehicles. Future work will further examine fault finding effectiveness of context-aware test generation.

References

[1] S. Piparia, M. K. Khan, and R. Bryce, "Discovery of real world context event patterns for smartphone devices using conditional random fields," in *ITNG 2021 18th International Conference on Information Technology-New Generations*, S. Latifi, Ed. Cham: Springer International Publishing, 2021, pp. 221–227.

[2] Google Inc, "Ui/application exerciser monkey." [Online]. Available: http://developer.android.com/tools/help/monkey.html Accessed 26 Dec 2021.

[3] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 224–234.

[4] D. Amalfitano, N. Amatucci, A. R. Fasolino, and P. Tramontana, "A Conceptual Framework for the Comparison of Fully Automated GUI Testing Techniques," in *International Conference on Automated Software Engineering Workshop (ASEW)*, 2015, pp. 50–57.

[5] A. Jaaskelainen, M. Katara, A. Kervinen, M. Maunumaa, T. Paakkonen, T. Takala, and H. Virtanen, "Automatic gui test generation for smartphone applications-an evaluation," in *International Conference on Software Engineering-Companion Volume*. IEEE, 2009, pp. 112–122.

[6] A. Nieminen, A. Jaaskelainen, H. Virtanen, and M. Katara, "A comparison of test generation algorithms for testing application interactions," in *Intl. Conference on Quality Software*. IEEE, 2011, pp. 131–140.

[7] D. Amalfitano, A. R. Fasolino, P. Tramontana, and N. Amatucci, "Considering context events in event-based testing of mobile applications," in *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2013, pp. 126–133.

[8] T. Griebe and V. Gruhn, "A model-based approach to test automation for context-aware mobile applications," in *ACM Symposium on Applied Computing*. ACM, 2014, pp. 420–427.

[9] Z. Liu, X. Gao, and X. Long, "Adaptive random testing of mobile application," in *International Conference on Computer Engineering and Technology (ICCET)*, vol. 2. IEEE, 2010, pp. V2–297.

[10] K. Song, A. R. Han, S. Jeong, and S. Cha, "Generating various contexts from permissions for testing android applications," in *Software Engineering and Knowledge Engineering (SEKE)*, 2015, pp. 87–92.

[11] C. Q. Adamsen, G. Mezzetti, and A. Møller, "Systematic execution of android test suites in adverse conditions," in *International Symposium on Software Testing and Analysis*. ACM, 2015, pp. 83–93.

[12] T. A. Majchrzak and M. Schulte, "Context-dependent testing of applications for mobile devices," *Open Journal of Web Technologies (OJWT)*, vol. 2, no. 1, pp. 27–39, 2015.

[13] T. Griebe, M. Hesenius, and V. Gruhn, "Towards automated UI-tests for sensor-based mobile applications," in *Intl. Conf. on Intelligent Software Methodologies, Tools, and Techniques*. Springer, 2015, pp. 3–17.

[14] Uber, "Calabash-android," 2019, retrieved Feb 25, 2020 from https://github.com/calabash.

[15] C.-J. M. Liang, N. D. Lane, N. Brouwers, L. Zhang, B. F. Karlsson, H. Liu, Y. Liu, J. Tang, X. Shan, R. Chandra, and F. Zhao, "Caiipa: Automated large-scale mobile app testing through contextual fuzzing," in *Intl. Conf. on Mobile Computing and Networking*. New York, NY, USA: ACM, 2014, pp. 519–530.

[16] G. Hu, X. Yuan, Y. Tang, and J. Yang, "Efficiently, effectively detecting mobile app bugs with appdoctor," in *Proceedings of the Ninth European Conference on Computer Systems*, 2014, pp. 1–15.

[17] 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*, ser. MOBILESoft '16. New York, NY, USA: ACM, 2016, pp. 88–99.

[18] A. S. Ami, M. M. Hasan, M. R. Rahman, and K. Sakib, "Mobicomonkey - context testing of android apps," in *Intl. Conf. on Mobile Software Engineering and Systems (MOBILESoft)*, May 2018, pp. 76–79.

[19] L. R. Rabiner, "A tutorial on hidden markov models and selected applications in speech recognition," *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257–286, Feb 1989.

[20] D. Cutting, J. Kupiec, J. Pedersen, and P. Sibun, "A practical part-of-speech tagger," in *In Proceedings of the Third Conference on Applied Natural Language Processing*, 1992, pp. 133–140.

[21] C. Sutton and A. McCallum, "An introduction to conditional random fields," *Foundations and Trends® in Machine Learning*, vol. 4, no. 4, pp. 267–373, 2012. [Online]. Available: http://dx.doi.org/10.1561/2200000013

[22] D. Adamo, D. Nurmuradov, S. Piparia, and R. Bryce, "Combinatorial-based event sequence testing of android applications," *Information and Software Technology*, vol. 99, pp. 98–117, 2018.

[23] S. Piparia, D. Adamo, R. Bryce, H. Do, and B. Bryant, "Combinatorial testing of context aware android applications," in *2021 16th Conference on Computer Science and Intelligence Systems (FedCSIS)*. IEEE, 2021, pp. 17–26.

[24] F-Droid, "F-droid: Free and open source android app repository," http://f-droid.org, 2017, (Accessed: 26-12-2021).

[25] Mountainminds GmbH, "EclEmma: JaCoCo java code coverage library," http://www.eclemma.org/jacoco/, 2017, (Accessed: 26-12-2021).