

# Combinatorial Testing of Context Aware Android Applications

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

David Adamo  
 Square, Inc.  
 dadamo@squareup.com

Renee Bryce, Hyunsook Do, Barrett Bryant  
 University of North Texas  
 {Renee.Bryce, Hyunsook.Do, Barrett.Bryant}@unt.edu

**Abstract**—Mobile devices such as smart phones and smart watches utilize apps that run in context aware environments and must respond to context changes such as changes in network connectivity, battery level, screen orientation, and more. The large number of GUI events and context events often complicate the testing process. This work expands the AutoDroid tool to automatically generate tests that are guided by PairwiseInterleaved coverage of GUI event and context event sequences. We systematically weave context and GUI events into testing using the pairwise interleaved algorithm. The results show that the pairwise interleaved algorithm achieves up to five times higher code coverage compared to a technique that generates test suites in a single predefined context (without interleaving context and GUI events), a technique that changes the context at the beginning of each test case (without interleaving context and GUI events), and Monkey-Context-GUI (which randomly chooses context and GUI events). Future work will expand this strategy to include more context variables and test emerging technologies such as IoT and autonomous vehicles.

**Index Terms**—Context, Android, EDSS, Combinatorial

## I. INTRODUCTION

SMART phone applications are becoming increasingly prevalent nowadays with widespread adoption. Graphical User Interface (GUI) applications have more complex structures to process a wide variety of context data pertaining to different scenarios compared to conventional computer software. This makes it more difficult for testing since they tend to have a rich selection of features. Due to infinite event combinations and fragmentation of supported devices for GUI applications, it is challenging in terms of time and costs to test them. Smart phone context-aware applications are emerging in multiple domains after integration of small-scale micro electromechanical sensors. For example, the Uber [1] smart phone application uses location services to help passengers to purchase rides through their technology that is able to access a customer’s location and nearby available vehicles and drivers. The app relies on user events to request service and context events, i.e., the Global Position System (GPS) reports their location. Some applications are based on advance machine learning algorithms such as CarSafe [2], a mobile phone application which uses driver behavior and road conditions to enable safe driving. This application uses the front camera to monitor the driver and rear camera to monitor road conditions when the smartphone is mounted on the car windscreen. The application also makes use of GPS,

gyroscope, and accelerometer to infer the vehicle’s movement and triggers alerts if dangerous driving behavior is detected. Incorporation of such context events further complicates the testing process.

Although mobile context aware applications has made our environment easier and intelligent, the complexity of such applications poses a challenge for testing. Researchers have proposed numerous techniques to simulate testing of context-aware mobile applications. Designing such methods is highly challenging because of the following reasons:

- Eco system fragmentation: Different mobile platforms often have unique architectures and features which make it difficult to generalize context simulation. Certain actions supported on one platform may not work for another platform which leads to platform-specific simulation of context events.
- Context heterogeneity: A wide variety of sensors are available which simulate context-aware behavior of mobile applications. Each sensor type may need a different mechanism suitable for simulation.
- Platform limitations: The Integrated Development Environments (IDEs) provide an easy simulation of Graphical User Interface (GUI) events on emulators but less support for simulation of context events.
- Budgets: Typically, mobile context-aware applications rely on various contextual input sources in addition to GUI events which make it infeasible to exhaustively test all input combinations.

This work presents a black-box approach to automatically generate test cases for context-sensitive Android applications. We expand the Autodroid tool [3] to focus on context-aware applications that are affected by four specific context variables: internet, power, battery, and screen orientation. The major components of Autodroid include the test builder, abstraction manager (which uses Appium to identify GUI events in different states of Application Under Test (AUT)), event selector, and event executor. The Autodroid tool is designed to automatically generate test suites with test cases of varying lengths. Each test case can be re-executed in isolation which makes it easier for testers to understand the scenarios and failure reproduction. Our context simulation provides a way to reduce time and cost and improves quality of testing. We

use a pair-wise strategy to simulate context and GUI event combinations. We compare our test suites obtained from extended Autodroid compared our results with Monkey-Context-GUI tool. Results indicate that our pairwise strategy improves code coverage in comparison to Monkey-Context-GUI and NoContext for five applications and improves code coverage in comparison to ISContext for three out of five applications.

## II. BACKGROUND AND RELATED WORK

This section provides background information about context aware systems and testing.

### A. Background

**Context aware applications:** A context event may affect the manner in which the mobile application responds to subsequent user interaction events. For example, when a button element is clicked, a mobile application that needs to respond by downloading a file from internet may exhibit different behaviors depending on whether or not the preceding context event indicated a switch into airplane mode. Context based applications may respond to a context event usually by some sort of change in state. This change in state may be physical and observable or may be logical and un-observable. For instance, a mobile application may reduce the rate at which it sends data over a network when battery levels go below a certain threshold. Android implements a BroadcastReceiver to listens for such context events. We formally define a context event in Definition 1.

**Definition 1.** A context event is a set of context variables. A context variable is a 2-tuple  $(c, a)$  where  $c$  is a context category and  $a$  is a context action.

WiFi	Battery	AC Power	Screen Orientation
Connected	Ok	Connected	Portrait
Disconnected	Low	Disconnected	Landscape
-	High	-	-

TABLE I: Combinatorial testing model with four context variables and different values for each variable

For example, Table I shows a combinatorial context model with four context variables (WiFi, battery, AC power and screen orientation) with two possible values for each context variable except battery which has three. We define a context event as combination of various values of these variables i.e.  $c = \{WiFi = connected, Battery = high, Power = connected, ScreenOrientation = portrait\}$ .

Android applications are Event Driven Systems (EDS) but unlike other traditional applications like web applications, they are more likely to sense and react to different kinds of events. These events could be generated by system itself or other applications. This section discusses previous studies which provide methods to test context-aware mobile applications.

### B. Related Work

1) *Online GUI testing:* Many tools and techniques for automated GUI testing of mobile applications exist [4]–[12]. The majority of these tools do not consider context changes and potential interactions between context variables during test generation. Test suites generated in a single predefined context may explore only the subset of GUI states and code that is reachable in a predefined context.

Machiry et al. [5] consider an application as an event-driven program that primarily interacts with its environment using a sequence of events via the Android Framework, called Dynodroid. Dynodroid can observe the reaction of the application upon each event while employing it as a guide for the generation of the next event. In addition, Dynodroid permits the interleaving of events generated by machines (numerous inputs) with the events generated by humans (intelligent events). They examined the capability of Dynodroid on 50 open-source applications while comparing the results with the same obtained via manually exercising applications and Monkey. Their study demonstrated that while Dynodroid covered lesser Java source code when compared with human approach, Dynodroid was still better than Monkey. Furthermore, Monkey took approximately 20 times more events than Dynodroid.

Amalfitano et al. [13] discuss GUI test automation using algorithms that traverse GUIs through continuous interaction and exploration. The algorithms simultaneously define and run test cases during the execution of an application. They use a generalized parametric algorithm to extract key aspects of the testing techniques while delivering a framework that can be employed to define and compare these testing techniques. Autodroid uses online GUI testing techniques similar to the ones mentioned by Amalfitano et al. [13] and Dynodroid.

2) *Context based GUI testing:* Test cases that only consider GUI events reduces the likelihood of finding faults that are only triggered by changes in context. This has been acknowledged in mobile application testing research [5], [14]–[17]. Existing work has built mobile application GUI testing tools that consider context events and GUI events. One such tool is Dynodroid [5]. Dynodroid generates a sequence of GUI and context events that is then fed as input to a mobile application under test. Each individual event is added to the input sequence using a random selection strategy. Dynodroid does not offer a way to systematically reduce and explore the range of possible context and GUI event combinations.

Song et al. propose an alternative approach to testing context-sensitive behavior of mobile applications [17]. The demonstrated approach tests context-sensitive behavior by executing a test suite multiple times in different contexts. Executing a test suite in multiple contexts can result in a situation where other valid test cases become infeasible in contexts different from that in which they were generated. Also, the approach is not cost-effective since the number of test cases to be executed increases significantly with the number of contexts to be tested. Furthermore, the proposed approach does not consider the use or impact of context event

sequences interleaved with GUI events. Adamsen et al. [18] and Majchrzak et al. [19] describe similar techniques for augmenting preexisting test suites with context information. Techniques that modify preexisting test suites or re-execute them in multiple contexts may cause test cases to become infeasible. This is because introduction of context events after test generation may cause the AUT's behavior to differ from the expected behavior in the preexisting test suites.

Amalfitano et al. [14] introduce interleaving context event sequences with GUI events during mobile application test generation. They adopt an event-patterns-based testing approach with sequences of context events that may be used to test a mobile application. The work demonstrates the benefits of using such event patterns for testing mobile applications. The experiments were carried out using a small set of manually and arbitrarily defined event patterns. Future work may create an event-pattern repository.

Griebe et al. [15] describe a model-based technique for automated testing of context-aware mobile applications. The technique requires manual creation of annotated UML Activity Diagrams that describe the behavior and context parameters of the AUT. The authors present an approach to incorporate sensor input for user acceptance tests [20]. This approach can generate sensor values as test cases by extending a UI testing tool, Calabash-Android [21]. To provide higher abstraction in test cases, this approach parses human language expressions (e.g. I shake the phone) to generate data using a mathematical model. The sensor data obtained can be used in test cases which are written in Gherkin language. Moran et al. [22] develop a tool called Crashescope that uses information from static analysis of source code to test contextual features in Android apps. Amalfitano et al. [14] propose a technique that requires manual specification of context event patterns that can be included in Android application test cases. The authors do not describe a systematic way to interleave context events with GUI events during test generation and do not consider potential interactions between context variables.

Liang et al. [23] present a cloud based service, CAIIPA, for testing context aware mobile applications. This study uses combinations of real world context events and improves crash and performance bug detection by 11.1x and 8.4x as compared to Monkey-Context-GUI-testing when evaluated on 265 Windows applications. While CAIIPA [23] utilizes real-world data for emulation of context events, the context events are limited to coarse-grained hardware parameters such as WiFi network, CPU device memory and sensor input and lacks on contexts such as screen orientation. Also, CAIIPA focuses only on Windows applications. Hu et al. [24] propose a cloud-based automation tool known as AppDoctor which injects actions such as network states, GUI gestures, intents, and the changes of device storage into an app to explore its possible executions.

Gomez et al. [25] describes a tool MoTiF, a crowd based approach to reproduce context sensitive crashes for Android applications. Crash patterns are generated from subject applications and is modified for application under test.

This technique can successfully reproduce crashes but crash patterns may miss some information which may not be generally applicable across applications. Ami et al. proposes MobiCoMonkey [26] which allows for contextual testing of Android applications. MobiCoMonkey utilizes the tool offered by Android SDK (Monkey) by associating context events with the tool. The context events are either introduced randomly by the tool or could be predefined by users. However, some GUI events are affected only by certain context events and randomly firing context events might miss on such dependencies which needs human intervention to be avoided. Also, MobiCoMonkey does not offer a systematic way to combine GUI and context events.

### III. CONTEXT MODELING AND PAIRWISE EVENT SELECTION STRATEGY

Our framework takes a context model as input that contains a set of context variables and possible values for each variable in order to generate a pairwise covering array. There are 24 possible value combinations for context events shown in Table I. It is possible to expand the combinatorial context model to include other variable (e.g. bluetooth, GPS, etc.). Changing the operating context of an application for exhaustive combination of context variables makes it computationally expensive since the number of combinations increases with the number of context variables. A  $t$ -way covering array can be used to model the operating context of an AUT. For a combinatorial model with  $k$  variables and  $v$  possible values for each variable, a  $t$ -way covering array  $CA(N; t; k; v)$  has  $N$  rows and  $k$  columns such that each  $t$ -tuple occurs at least once within the rows, where  $t$  is the strength of interaction coverage [27].

ID	WiFi	Battery	AC Power	Screen Orientation
$c_1$	Disconnected	Low	Disconnected	Landscape
$c_2$	Connected	Low	Connected	Portrait
$c_3$	Disconnected	Okay	Connected	Landscape
$c_4$	Connected	Okay	Disconnected	Portrait
$c_5$	Disconnected	High	Disconnected	Portrait
$c_6$	Connected	High	Connected	Landscape

TABLE II: A 2-way covering array that defines six contexts

Table II shows a pairwise covering array for the combinatorial context model in Table I. To assess the functionality of the AUT, each operating context is modified as represented by the row of the covering array  $c_i$ . The CATDroid framework generates covering arrays to test interactions between a subset of context variables.

### IV. TEST SUITE CONSTRUCTION FRAMEWORK

The CATDroid framework uses an *event extraction cycle* to iteratively select and execute events from the GUI of the application under test and to construct test cases one-event-at-a-time.

Figure 1 shows pseudo code for the test suite construction framework to automatically construct test suites for context sensitive Android apps. The algorithm requires input: (i) an Android Application Package (APK) file, (ii) context events

```

Input: android application package, AUT
Input: combinatorial context model, M
Input: initial context selection strategy, initialContextSelection
Input: event selection strategy, eventSelection
Input: test case termination criterion, terminationCriterion
Input: test suite completion criterion, completionCriterion
Output: test suite, T
1:  $C_{all} \leftarrow$  generate context covering array from M
2:  $T \leftarrow \phi$  ▷ test suite
3: repeat
4:    $t_i \leftarrow \phi$  ▷ test case
5:    $C_{curr} \leftarrow$  InitialContextStrategy( $C_{all}$ )
6:   add initial context event,  $C_{curr}$  to test case  $t_i$ 
7:   install and launch AUT, add launch event to  $t_i$ 
8:    $s_{curr} \leftarrow$  initial GUI state
9:   while TerminationCriterion is not satisfied do
10:     $E_{all} \leftarrow$  GUI events in current GUI state  $s_{curr}$ 
11:     $e_{sel} \leftarrow$  EventSelectionStrategy( $s_{curr}, E_{all}, C_{all}$ )
12:    execute  $e_{sel}$ 
13:     $t_i \leftarrow t_i \cup \{e_{sel}\}$ 
14:     $s_{curr} \leftarrow$  current GUI state
15:  end while
16:   $T \leftarrow T \cup \{t_i\}$ 
17:  finalize test case (clear cache/SD card, uninstall app, etc.)
18: until CompletionCriterion is satisfied

```

Fig. 1: Pseudocode for the extended Autodroid framework (boxes indicate framework parameters)

needed for a combinatorial model, (iii) an initial context strategy, (iv) a test case termination criterion, and (v) test suite completion criterion. The test case termination criterion terminates the sequences of events either on the basis of the length of each sequence or probability. The test suite completion criterion may be predefined number of test cases or a fixed time. Lines 9-15 represent the event extraction cycle that incrementally constructs each test case. The framework requires specifications for several parameters (shown in boxes) to instantiate different test generation techniques. The algorithm includes four steps:

**Step 1: Generate context covering array.** Line 1 generates a covering array  $C$  using [28] from the combinatorial context model  $M$  specified as input. The covering array specifies a set of context events that will be used to test the AUT. Each context event specified in the covering array has a corresponding set of context variables that changes the operating context of the AUT. The generated set of context events is used to set initial context at the beginning of the test case as well as for the pairwise event selection strategy.

**Step 2: Initialize test case.** Lines 4-8 initialize each test case in the test suite. Line 4 creates an empty event sequence. Line 5 uses a predefined strategy to iterate over a context covering array  $C_{all}$  and selects a different context event at the

```

Input: current GUI state,  $s_{curr}$ 
Input: GUI events in current state,  $E_{all}$ 
Input: covering array generated from pairwise combinations of context events,  $C_{all}$ 
Input: set of covered context-GUI pairs, Context-GUI-pairs
Input: set of covered context-state pairs, Context-state-pairs
Output: GUI event or context event,  $e_{sel}$ 
1:  $c_{curr} \leftarrow$  get_current_emulator_context()
2:  $e_{sel} \leftarrow$  select a GUI event  $e_i \in E_{all}$  such that  $(c_{curr}, e_i)$  is not in Context-GUI-pairs
3: if  $e_{sel}$  is NULL then
4:   Context-state-pairs  $\leftarrow (c_{curr}, s_{curr})$ 
5:    $c_{sel} \leftarrow$  select a context  $c_i \in C_{all}$  such that  $(c_i, s_{curr})$  is not in Context-state-pairs
6:   if  $c_{sel}$  is not NULL then
7:      $e_{sel} \leftarrow c_{sel}$ 
8:   return  $e_{sel}$ 
9: else
10:    $e_{sel} \leftarrow$  select a GUI event  $e_i \in E_{all}$  randomly
11: end if
12: end if
13: Context-GUI-pairs  $\leftarrow (c_{curr}, e_{sel})$ 
14: return  $e_{sel}$ 

```

Fig. 2: Pseudocode for Pairwise Algorithm

beginning of each test case. Line 7 launches the AUT in the selected start context event and adds a launch event to the test case. Line 8 retrieves the initial GUI state of the AUT.

**Step 3: Select and execute an event.** The *EventSelectionStrategy* procedure call on line 11 uses a predefined strategy to select and execute a context event or GUI event in each iteration of the event extraction cycle (lines 9-15). Event execution often changes the GUI state of the AUT and/or the value of one or more context variables. This iterative event selection and execution incrementally constructs a test case that may include context events and GUI events. In each iteration of the event extraction cycle, the *EventSelectionStrategy* parameter specifies a strategy to choose (i) whether to execute a GUI event or context event and (ii) which particular event to execute given a set of available GUI events, and a context covering array. Figure 2 describes the technique to interleave context and GUI events. A single test case ends when the algorithm satisfies a predefined *TerminationCriterion*.

**Step 4: Finalize test case.** At the end of each test case, line 17 resets the state of the AUT and clears all data that may affect the outcome of subsequent test cases.

The algorithm generates multiple test cases until it satisfies the *CompletionCriterion* that specifies when the test suite is complete.

**Pairwise event selection procedure.** Figure 2 describes the *EventSelectionStrategy* to select either a context or a GUI event prior to adding an event in the test case. The algorithm requires the input: (i) current GUI state, (ii) GUI events available in

current state and (iii) context covering array, (iv) the set of covered context-GUI event pairs, and (v) the set of covered context-state pairs. The event selection occurs at line 11 of Figure 1. The algorithm tracks coverage of context-GUI pairs and context-state pairs at the test suite level. Context-GUI pairs keep track of GUI events executed in a particular context which ensures that all possible GUI events are executed in a particular context before changing the context of the emulator. Context-state pairs keep track of context state change in a GUI state. We start by fetching the current context state of the emulator at line 1. A GUI event is selected from available events that has not been executed yet in the current context. If such a GUI event exists, the GUI event is selected and the context-GUI pair is marked as covered. Once all GUI events in a current context are covered, the algorithm changes the context by selecting a value from covering array indicated in lines 3-12 such that there is at least one GUI event in the current GUI state that has not yet been executed in the chosen context. This step occurs at line 5. All tie breaks are performed randomly. We consider a total of 6 contexts based on the 2-way covering array in Figure II and depending on the GUI events for the AUT, different context-to-GUI ratio may exist for different applications. When all context-state pairs and context-GUI pairs are covered in a particular GUI state, the algorithm randomly selects a GUI event.

App Name	Installations	Version	Lines	Methods	Classes
Diode	10,000+	1.3.2.2	7933	1134	209
BartRunner	100,000+	2.2.19	3644	750	135
Your Local Weather	5000+	5.6.4	15062	499	114
MovieDB	1000+	2.1.1	2719	319	81
Abcore	1000+	0.77	1215	197	46

TABLE III: Characteristics of selected Android apps

## V. EXPERIMENTAL STUDY

This section presents results of an empirical study with five applications with characteristics mentioned in Table III.

### A. Experimental Setup

We use Android 10.0 Pixel emulator with API 29 and generate 10 test suites with each technique for five applications chosen from F-droid [29] with a total of 200 test suites. These subject applications are instrumented with JaCoCo [30]. We use a fixed probability value of 0.05 to terminate a test case with a two second delay between execution of events for each test case so that the AUT can respond to each event. We set a fixed time budget of two hours for test suite completion. We compare the code coverage for test suites obtained from various techniques mentioned in Section V-B.

### B. Variables and Measures

This section discusses the variables and metrics used in our experiments.

**Independent Variable** Our independent variable is test generation technique and we consider three controls (Monkey-Context-GUI, NoContext, and ISContext) and one heuristic (PairwiseInterleaved) as follows:

- The **NoContext** technique generates a test suite by executing GUI events without consideration for context changes using our CATDroid tool. We used the *NoContext* technique to construct test suites in a single context  $c = \{WiFi=connected, Battery=OK, AC Power=connected, ScreenOrientation=Portrait\}$  that represents favorable operating conditions for the AUT.
- The **IterativeStartContext (ISContext)** technique selects a different context event at the beginning of each test case by iterating through the context covering array in a round-robin manner. After choosing a context, the technique makes a random selection among GUI events.
- The **PairwiseInterleaved** technique selects a different context event at the beginning of each test case by iterating through the context covering array in a round-robin manner using our CATDroid tool. It also systematically interleaves context events with GUI events by prioritizing the execution of GUI events in new contexts as described in Figure 2.
- **Monkey-Context-GUI** [31] takes a predefined number of events as input. It executes an action on the GUI application by performing clicks randomly on the screen coordinates regardless of whether the events are relevant to the application under test (AUT). Monkey-Context-GUI generates a single event sequence for each test suite. We configure Monkey-Context-GUI to generate multiple event sequences for each test suite of 120 minutes. For each application, we find the maximum event sequence length across test suites for NoContext, ISContext, and PairwiseInterleaved techniques and provide it as input to Monkey-Context-GUI. Monkey-Context-GUI executes context events randomly without knowledge of the GUI events for the AUT. We considered Monkey-Context-GUI as a baseline for evaluation of test suites obtained using CATDroid since it is one of the few existing tools that is compatible with recent versions of Android OS.

**Dependent Variables** We use the following code coverage metrics to investigate our research questions:

- **Line coverage** Line coverage measures the total number of covered source code statements.
- **Method coverage** Method coverage indicates whether a method was entered at all during execution.
- **Class coverage** Class coverage metric measures how many classes were executed by a test suite.

### C. Research Questions

Our experiments address the following research questions:

- RQ1:** Does the PairwiseInterleaved technique increase line, method, and class coverage compared to Monkey-Context-GUI, NoContext, and ISContext?

Application	Monkey-Context-GUI			NoContext			ISContext			PairwiseInterleaved		
	Line Coverage	Method Coverage	Class Coverage	Line Coverage	Method Coverage	Class Coverage	Line Coverage	Method Coverage	Class Coverage	Line Coverage	Method Coverage	Class Coverage
Diode	6.29	11.56	9.28	32.44	43.81	38.42	33.33	45.37	40.08	34.15	45.83	40.24
Abcore	5.51	5.58	4.35	15.83	25.38	21.74	<b>58.63</b>	65.84	67.61	58.03	<b>65.94</b>	<b>68.48</b>
MovieDB	6.97	11.41	13.58	40.65	47.36	51.36	49.45	57.58	60.62	<b>52.71</b>	<b>59.56</b>	<b>63.58</b>
YourLocalWeather	4.73	8.31	16.66	9.04	15.31	27.92	9.19	15.44	<b>27.93</b>	<b>9.22</b>	<b>15.51</b>	26.74
BartRunner	10.31	12.43	20.96	<b>59.06</b>	<b>62.93</b>	<b>75.41</b>	58.09	61.60	<b>75.41</b>	51.94	55.77	70.89
<b>Average</b>	6.76	9.86	12.97	31.41	38.96	42.97	41.74	49.16	54.33	41.21	48.52	53.98

TABLE IV: Average code coverage for Monkey-Context-GUI, NoContext and PairwiseInterleaved test suites

**RQ2:** Do control techniques (NoContext, ISContext, and Monkey-Context-GUI) perform differently in terms of line, method, and class coverage?

#### D. Results and Analysis

Table IV shows the average line, method, and class coverage across ten runs of our techniques, Monkey-Context-GUI, NoContext, ISContext, and PairwiseInterleaved for each subject application. The values in bold indicate the highest values of line, method, and class coverage across all techniques for five subject applications. The bottom row shows the average values for each approach (i.e. Monkey-Context-GUI, NoContext, ISContext, and PairwiseInterleaved techniques) although the results vary across applications as discussed in Section V-E.

Application	NoContext over Monkey-Context-GUI		
	Line Coverage	Method Coverage	Class Coverage
Diode	5.16	3.78	4.13
Abcore	2.87	4.54	<b>4.99</b>
MovieDB	<b>5.83</b>	4.14	3.78
YourLocalWeather	<i>1.91</i>	<i>1.84</i>	<i>1.67</i>
BartRunner	5.72	<b>5.06</b>	3.59
<b>Average</b>	4.30	3.88	3.64

TABLE V: Ratio for NoContext over Monkey-Context-GUI

**RQ1 Results:** The PairwiseInterleaved technique shows twice as much line coverage when compared to Monkey-Context-GUI for the application *Your Local Weather*. On the other hand, the PairwiseInterleaved technique obtains significant improvements (up to ten times) in terms of line coverage for the application *Abcore*. Similarly, the test suites obtained from PairwiseInterleaved technique for four subject applications showed improvements in terms of line coverage over Monkey-Context-GUI. The method coverage and class coverage follows a similar pattern for the PairwiseInterleaved technique when compared to Monkey-Context-GUI.

For the application *Bart Runner*, the NoContext and ISContext techniques outperformed the PairwiseInterleaved technique in terms of line coverage. The applications *Your Local Weather* and *Diode* show similar line coverage for test suites obtained from the NoContext, ISContext, and PairwiseInterleaved technique. However, for the application *Movie DB*, a considerable improvement in line coverage was observed for the PairwiseInterleaved technique over the NoContext and ISContext techniques. We notice similar behavior for both techniques in terms of method and class coverage.

Our PairwiseInterleaved technique often achieved 6.1 times (absolute difference of 34.45%), 4.9 times (absolute difference of 38.66%), and 4.2 times (absolute difference of 41.02%) higher line, method, and class coverage in comparison to

Monkey-Context-GUI. It also achieved 1.31 times (absolute difference of 9.8%), 1.25 times (absolute difference of 9.7%), and 1.26 times (absolute difference of 11.01%) higher line, method, and class coverage in comparison to NoContext. However, the PairwiseInterleaved technique does not show an improvement over ISContext on an average. This is because the average value does not compare the impact of individual application. We notice that PairwiseInterleaved technique indicates better or comparable results for three out of five applications with an average of 1.37% improvement in terms of line, method, and class coverage.

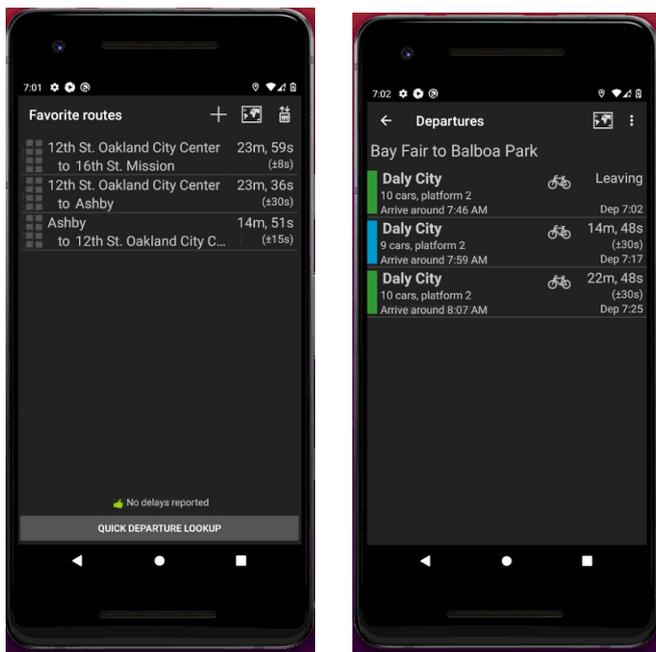
**RQ2 Results:** Table IV shows that line, method, and class coverage for the ISContext technique offers little improvement in comparison to the NoContext technique for *Your Local Weather* and *Diode* applications. The improvement in line coverage for ISContext over the NoContext technique was highest for the application *Abcore* (by a factor of 3.7). The application *MovieDB* showed an improvement in line and method coverage for ISContext over NoContext by a factor of 1.21. The class coverage showed 1.18 times improvement for *MovieDB*. The application *Bart Runner* performs better in terms of line and method coverage for NoContext when compared to ISContext by a small margin (by 1% and 1.3% respectively). The class coverage is same for both techniques. On an average, the ISContext technique achieved 1.32 times (10.32%), 1.26 times (10.2%), and 1.26 times (11.35%) higher line, method, and class coverage in comparison to NoContext.

Table IV indicates that ISContext achieves a higher line, method, and class coverage than Monkey-Context-GUI for all five applications. The application *Abcore* achieves 10.64 times line coverage, 11.79 times method coverage, and 15.54 times class coverage for ISContext technique over Monkey-Context-GUI. This is the highest improvement obtained between any two techniques across all subject applications. The applications *BartRunner* and *Diode* show approximately 5 times improvement in line coverage for ISContext in comparison to Monkey-Context-GUI. The method and class coverage follow a similar pattern. The ISContext technique for the *MovieDB* application show 7 times line coverage, 5 times method coverage and 4.4 times class coverage when compared to Monkey-Context-GUI. The application *Your Local Weather* shows an improvement for ISContext over Monkey-Context-GUI by a factor of 1.94 times for line coverage, 1.85 times for method coverage, and 1.7 times for class coverage. On an average, ISContext technique achieved 6.17 times (absolute difference of 41.74%) higher line coverage, 5 times (absolute difference of 49.16%) higher method coverage, and 4.2 times (absolute difference of 54.33%) higher class coverage when compared to Monkey-Context-GUI.

Table IV shows that line, method, and class coverage for the NoContext technique showed improvements over the Monkey-Context-GUI for *Your Local Weather* application (by a factor of more than 1.5). Likewise, the improvement in line coverage for the NoContext technique was highest for the application *Movie DB* whereas *Bart Runner* and *Abcore* showed the highest improvement in method and class coverage, respectively. An average, the NoContext technique achieved 4.6 times (absolute difference of 24.64%), 4 times (absolute difference of 29.10%), and 3.3 times (absolute difference of 30%) higher line, method, and class coverage in comparison to Monkey-Context-GUI.

Application	ISContext over NoContext		
	Line Coverage	Method Coverage	Class Coverage
Diode	1.03	1.04	1.04
Abcore	<b>3.70</b>	<b>2.59</b>	<b>3.11</b>
MovieDB	1.22	1.22	1.18
YourLocalWeather	1.02	1.01	<i>1.00</i>
BartRunner	<i>0.98</i>	<i>0.98</i>	<i>1.00</i>
<b>Average</b>	1.33	1.26	1.26

TABLE VI: Ratio for ISContext over NoContext



(a) Bart Runner Routes

(b) Bart Runner Departures

Fig. 3: The Bart Runner application

### E. Discussion and Implications

To understand the performance improvement of NoContext over Monkey-Context-GUI, we calculated their ratios (NoContext:Monkey-Context-GUI) of coverage for all subject applications, as shown in Table V. The numbers in bold indicate the highest ratio and the italicized numbers indicate the lowest ratio obtained for various code coverage. The bottom row shows the average improvement of ratios of NoContext technique when compared to Monkey-Context-GUI across all

subject applications. On an average, NoContext achieves 4.3 times line coverage, 3.88 times method coverage, and 3.64 times class coverage with different values across applications. We can see that the application *Movie DB* shows the highest ratio of line coverage across all applications. In addition, the apps *Bart Runner* and *Abcore* show improvement in method and class coverage, respectively. *Your Local Weather* also shows an improvement, although not as significant as other four applications. Although, Monkey-Context-GUI has an advantage for *Your Local Weather* application over extended Autodroid because Monkey-Context-GUI can add a location by choosing coordinates from the map component. Although, it still did not add a location due to its random nature. This analyses show that the NoContext technique outperforms our baseline, Monkey-Context-GUI, for all subject applications even though Monkey-Context-GUI considers execution of context events. This is because Monkey-Context-GUI does not have any information regarding the events in AUT. The random clicks performed by Monkey-Context-GUI sometimes does not lead to any action performed and hence the tool is not able to explore the application after a certain point. It is important to have information about the GUI events and states of the AUT to enable proper testing of Android applications.

We calculate the ratios of code coverage for ISContext over NoContext in Table VI. The bold and italicized values highlight the highest and lowest ratios. The bottom row shows the average. The only app for which ISContext does not perform better than NoContext is *Bart Runner*. The app makes use of internet to download train schedules but the huge number of GUI events overshadows context events and hence results in a low coverage for ISContext by approx 1%. This indicates that for this application, including context events could result in decreased code coverage and is not an ideal candidate for context-aware testing. *Your Local Weather* and *Diode* show a marginal improvement of ISContext over NoContext. This is because of limited dependency of these apps on context events. The application *Movie DB* and *Abcore* shows substantial improvement (approx. 9% and 41%) in line coverage for ISContext over NoContext.

The PairwiseInterleaved technique does not perform well for *BartRunner* when compared to NoContext and ISContext. *Bart Runner* is a scheduling application for trains in the US. It allows users to enter their most traveled routes and provides real-time list of upcoming departures. This application depends only on the three context variables; internet, wake locker, and alarm, but has numerous GUI events related to train routes as shown in Figure 3a and departures in Figure 3b. This leads to a high GUI to context events ratio which results in low coverage for our PairwiseInterleaved technique. For ISContext technique, the results are similar since the overhead of context over GUI events is not too much. However, both PairwiseInterleaved and ISContext techniques were able to cover some of the branches due to the absence of internet, which is not covered by NoContext algorithm across all test suites. Figure 4a shows the missing catch block for the NoContext technique which is covered by our Pairwise as

```

String xml = null;
try {
    String url;
    if (ignoreDirection || params.getOrigin().ignoreRoutingDirection) {
        url = String.format(ETD_URL_NO_DIRECTION);
    }
    return null;
}
return realtimeDepartures;
} catch (MalformedURLException | UnsupportedEncodingException e) {
    throw new RuntimeException(e);
} catch (IOException e) {
    if (attemptNumber < MAX_ATTEMPTS - 1) {
        try {
            Log.w(Constants.TAG,
                "Attempt to contact server failed... retrying in 3s",
                e);
            Thread.sleep(3000);
        } catch (InterruptedException interrupt) {
            // Ignore... just go on to next attempt
        }
        return getDeparturesFromNetwork(params, attemptNumber + 1);
    } else {
        mException = new Exception("Could not contact BART system", e);
        return null;
    }
}
}

```

(a) NoContext missed the catch block

```

try {
    String url;
    if (ignoreDirection || params.getOrigin().ignoreRoutingDirection) {
        url = String.format(ETD_URL_NO_DIRECTION);
    }
    return realtimeDepartures;
} catch (MalformedURLException | UnsupportedEncodingException e) {
    throw new RuntimeException(e);
} catch (IOException e) {
    if (attemptNumber < MAX_ATTEMPTS - 1) {
        try {
            Log.w(Constants.TAG,
                "Attempt to contact server failed... retrying in 3s",
                e);
            Thread.sleep(3000);
        } catch (InterruptedException interrupt) {
            // Ignore... just go on to next attempt
        }
        return getDeparturesFromNetwork(params, attemptNumber + 1);
    } else {
        mException = new Exception("Could not contact BART system", e);
        return null;
    }
}
}

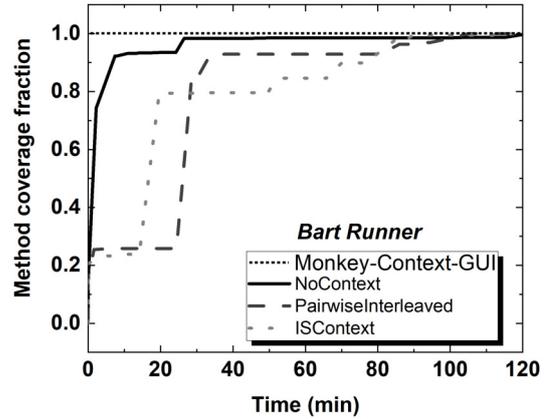
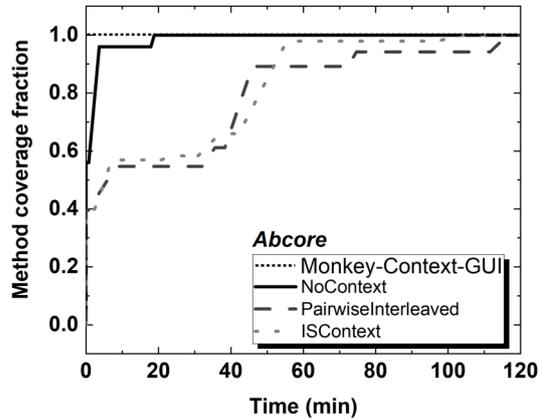
```

(b) PairwiseInterleaved and ISContext covers the catch block covered due to absence of Internet

Fig. 4: Code snippet for BartRunner for NoContext, ISContext, and PairwiseInterleaved techniques

well as ISContext techniques as shown in Figure 4b. *Bart Runner* uses two context events, alarm and wake locker, which were excluded from this study. Since applications are sensitive to specific context variables, it is important to have this information in advance and integrate those context variables in test case generation.

*Abcore* runs a Bitcoin core node on the Android device. This app starts by downloading data from WiFi and may not work well on 3G or 4G network. *Abcore* has very limited number of GUI events and high context events which lead to only 15% line coverage in case of NoContext. The coverage is significantly improved (by a factor of 3.7) when context is manipulated at the beginning of each test case as evident by values of the technique ISContext. Also, *Abcore* is the only application which includes explicit broadcasts for all four context variables (internet, power, battery, and WiFi). Given that NoContext technique explores the functionality of the AUT only under favorable conditions (esp. internet connected)

Fig. 5: Rate of method coverage for test suite with highest method coverage obtained for the application *Bart Runner*Fig. 6: Rate of method coverage for test suite with highest method coverage obtained for the application *Abcore*

for all GUI states, low code coverage is obtained due to lack in consideration of different values for these context variables. Due to these reasons, PairwiseInterleaved technique offers a large increase over the NoContext technique. The PairwiseInterleaved technique gave slightly lesser line coverage but slightly higher method and class coverage when compared to ISContext.

We also investigate the method coverage rate of Monkey-Context-GUI, NoContext, ISContext, and PairwiseInterleaved techniques for *Bart Runner* and *Abcore* applications. We plot the method coverage graph for the application *Bart Runner*. We chose method coverage here but line and class coverage follows a similar pattern. Figure 5 shows the method coverage rate for the test suite with maximum method coverage for Monkey-Context-GUI, NoContext, ISContext, and PairwiseInterleaved techniques. Here, the abscissa indicates the time and ordinate indicates the fraction of method coverage. This

fraction is obtained by normalizing the method coverage value by its maximum value. We plot this value with respect to time to obtain the method coverage rate. Similarly, we plot the method coverage graphs for the application *Abcore*. Figure 6 shows the rate of method coverage for test suites with maximum coverage. Monkey-Context-GUI reaches its maximum value from the beginning. NoContext reaches its maximum value rapidly for maximum coverage test suite when compared to ISContext and PairwiseInterleaved for both of these applications. The Monkey-Context-GUI and NoContext technique does not explore the AUT after it reaches its maximum value at a very early stage. The ISContext takes more time to reach its maximum value as compared to Monkey and NoContext but PairwiseInterleaved technique explores the AUT towards the later stage as well. Here, we show the graph of the test suite with maximum coverage but the test suite with minimum coverage also shows similar behavior.

Next, we analyze the results of the *Diode* and *Movie DB* applications due to their similar performances for PairwiseInterleaved technique when compared to ISContext and NoContext. *Diode* is a third party application which allows users to narrow the search for a particular reddit topic pertaining to a chosen theme. It features several choices for users which lead to numerous GUI events. *Movie DB* is an online application to explore database for movies and TV shows. NoContext explores approximately 40% of the *Movie DB* application and 32% of the *Diode* due to its huge number of GUI events. The ISContext technique shows 9% improvement in line coverage for the *Movie DB* when compared to NoContext. Similarly, ISContext shows 1% improvement in line coverage for the *Diode* application when compared to NoContext. This indicates that it is important to manipulate the context of the AUT during test generation. The PairwiseInterleaved technique slightly outperforms NoContext and ISContext for both of these applications. One reason for the favorable performance of ISContext and PairwiseInterleaved is that deeply nested GUI actions (i.e. actions far beyond the first screen) are affected by context. This is especially in case of the availability of an internet connection. So, despite the significant ratio of GUI events to context events there is a lot of variability in the outcome of a GUI event depending on internet availability or some other context. This indicates the importance of execution of GUI events in multiple contexts to test context-sensitive behavior. The overall code coverage for *Diode* application is also affected by the large number of topics available on reddit from which this application acquires its data.

The NoContext, ISContext, and PairwiseInterleaved techniques for *Your Local Weather* give similar results and show small improvement over Monkey-Context-GUI. *Your Local Weather* is a weather application that uses data network, WiFi, and GPS to show the weather of the current location. The application integrates a map component for users to specify a location. There are various reasons for the low code coverage for this application. The large size of this application in terms of lines of code made it difficult to explore the application within the allocated time frame. The map component is

missing from our tool and hence the application is not able to detect a location. None of the techniques fully explored the application which resulted in overall lack of coverage. Including GPS to detect a location may improve code coverage for this application. Furthermore, the application considers the context variable *DEVICE\_BOOT\_COMPLETE* to auto-start the application after the device is finished booting, which is excluded from our study.

#### F. Implications for Mobile Application Testing

GUI test case generation algorithms produce a sequence of events as test cases [5], [32], [33]. The behavior of a sequence of GUI events can vary depending on the current operating context of a mobile device. It is important to also generate events that manipulate the operating context of the device in addition to exercising the AUT using GUI events.

Figure 1 provides an algorithm to automatically generate test cases with interleaved sequences of context and GUI events. The algorithm uses a combinatorial model to define different contexts and uses the algorithm mentioned in Figure 2 to determine next likely context event. This helps to limit the decision space at each point of adding a context event to the test case.

Context-aware test cases have the potential to expose context-driven behavior that may otherwise go untested without context events. For instance, tapping a ‘download’ button in a mobile app may exhibit varying behavior depending on whether or not an internet connection is available. Context-aware test cases may even discover interactions between the operating context of the device and GUI events. However, the potential benefits of context-aware test cases depend on the nature of the AUT. If the AUT does not use the Internet in any way, changing the connectivity context of the device is unlikely to expose new behavior.

## VI. THREATS TO VALIDITY

This work applies the context driven testing strategies to five different applications that have different characteristics in terms of size, relevant context events, number of screens, and behaviors. The results may differ for applications with different characteristics. Another threat is the random nature of our techniques. To control this threat, we performed ten runs for test suite generation and reported the average values. One challenge for research in context-sensitive mobile app testing is the lack of reliable emulators for context events. This hinders automated generation of context-aware tests for mobile applications. The context variables in this study were limited to events possible to simulate in an Android emulator. A larger set of context variables and a different schedule for context event insertion may help achieve better results.

## VII. CONCLUSION AND FUTURE WORK

Smart phone applications are EDS which also react to context events that may cause its behavior to change. Context events may alter the operating context of an application under test which makes it important to generate tests that manipulate

the operating context of the AUT. This work provides a context aware automated testing framework (CATDroid) for automatic generation of test suites. We use our framework to instantiate multiple test case generation techniques that compares test case generation techniques with and without manipulating the context of AUT. The PairwiseInterleaved technique achieves higher line coverage up to a factor of six when compared to Monkey-Context-GUI, up to a 1.3 times increment in line coverage compared to a technique that generates test suites in a single predefined context, and achieves similar code coverage when compared to ISContext across all five subject applications. Our results indicate that the benefit of manipulating operating context and interleaving context events with GUI events depend on the characteristics of the AUT. Future work will explore a broader set of applications and context events and the impact of higher interaction strength coverage of context and GUI events.

#### REFERENCES

- [1] Uber, "Uber- earn money by driving or get a ride now," 2019, retrieved Feb 25, 2020 from <https://www.uber.com>.
- [2] C.-W. You, M. Montes-de Oca, T. J. Bao, N. D. Lane, H. Lu, G. Cardone, L. Torresani, and A. T. Campbell, "Carsafe: a driver safety app that detects dangerous driving behavior using dual-cameras on smartphones," in *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, 2012, pp. 671–672.
- [3] 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.
- [4] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "MobiGUITAR: Automated model-based testing of mobile apps," *IEEE Software*, vol. 32, no. 5, pp. 53–59, 2015.
- [5] 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.
- [6] I. C. Morgado and A. C. Paiva, "The iMPaCT tool: Testing UI patterns on mobile applications," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 876–881.
- [7] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek, "Reducing combinatorics in GUI testing of android applications," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 559–570.
- [8] D. Amalfitano, N. Amatucci, A. M. Memon, P. Tramontana, and A. R. Fasolino, "A general framework for comparing automatic testing techniques of android mobile apps," *Journal of Systems and Software*, vol. 125, pp. 322–343, 2017.
- [9] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 94–105.
- [10] R. Michaels, D. Adamo, and R. Bryce, "Combinatorial-based event sequences for reduction of android test suites," in *2020 10th Annual Computing and Communication Workshop and Conference (CCWC)*, 2020, pp. 0598–0605.
- [11] R. Michaels, M. K. Khan, and R. Bryce, "Mobile test suite generation via combinatorial sequences," in *ITNG 2021 18th International Conference on Information Technology-New Generations*, S. Latifi, Ed. Cham: Springer International Publishing, 2021, pp. 273–279.
- [12] 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.
- [13] D. Amalfitano, N. Amatucci, A. R. Fasolino, and P. Tramontana, "A Conceptual Framework for the Comparison of Fully Automated GUI Testing Techniques," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, 2015, pp. 50–57.
- [14] D. Amalfitano, A. R. Fasolino, P. Tramontana, and N. Amatucci, "Considering context events in event-based testing of mobile applications," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2013, pp. 126–133.
- [15] T. Griebe and V. Gruhn, "A model-based approach to test automation for context-aware mobile applications," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. ACM, 2014, pp. 420–427.
- [16] Z. Liu, X. Gao, and X. Long, "Adaptive random testing of mobile application," in *2010 2nd International Conference on Computer Engineering and Technology (ICCET)*, vol. 2. IEEE, 2010, pp. V2–297.
- [17] K. Song, A. R. Han, S. Jeong, and S. Cha, "Generating various contexts from permissions for testing android applications," in *27th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, 2015, pp. 87–92.
- [18] T. Q. Adamsen, G. Mezzetti, and A. Møller, "Systematic execution of android test suites in adverse conditions," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 2015, pp. 83–93.
- [19] 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.
- [20] T. Griebe, M. Hesenius, and V. Gruhn, "Towards automated UI-tests for sensor-based mobile applications," in *International Conference on Intelligent Software Methodologies, Tools, and Techniques*. Springer, 2015, pp. 3–17.
- [21] Uber, "Calabash-android," 2019, retrieved Feb 25, 2020 from <https://github.com/calabash>.
- [22] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshvanyk, "Automatically discovering, reporting and reproducing android application crashes," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2016, pp. 33–44.
- [23] 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, "Caijpa: Automated large-scale mobile app testing through contextual fuzzing," in *Proceedings of the 20th Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '14. New York, NY, USA: ACM, 2014, pp. 519–530. [Online]. Available: <http://doi.acm.org/10.1145/2639108.2639131>
- [24] 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.
- [25] 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. [Online]. Available: <http://doi.acm.org/10.1145/2897073.2897088>
- [26] A. S. Ami, M. M. Hasan, M. R. Rahman, and K. Sakib, "Mobicomonkey - context testing of android apps," in *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILE-Soft)*, May 2018, pp. 76–79.
- [27] R. C. Bryce and C. J. Colbourn, "Prioritized interaction testing for pairwise coverage with seeding and constraints," *Information and Software Technology*, vol. 48, no. 10, pp. 960–970, 2006.
- [28] H. Tsuyoshi. (2021) Pairwise combinatorial package. <https://github.com/thombashi/allpairspy>. (Accessed: 25-05-2021).
- [29] F-Droid, "F-droid: Free and open source android app repository," <http://f-droid.org>, 2017, (Accessed: 26-02-2021).
- [30] Mountainminds GmbH, "EclEmma: JaCoCo java code coverage library," <http://www.eclemma.org/jacoco/>, 2017, (Accessed: 26-02-2021).
- [31] Google, "UI/application exerciser monkey," 2017, retrieved May 10, 2021 from <https://developer.android.com/studio/test/monkey.html>.
- [32] A. M. Memon, "Developing testing techniques for event-driven pervasive computing applications," in *Proceedings of The OOPSLA 2004 workshop on Building Software for Pervasive Computing (BSPC 2004)*, 2004.
- [33] A. Memon, "An event-flow model of gui-based applications for testing," *Software testing, verification and reliability*, vol. 17, no. 3, pp. 137–157, 2007.