2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops

Considering Context Events in Event-Based Testing of Mobile Applications

Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Nicola Amatucci

domenico.amalfitano@unina.it, anna.fasolino@unina.it, porfirio.tramontana@unina.it, nicola.amatucci@gmail.com

Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione, Università di Napoli Federico II, Via Claudio 21, 80125 Napoli, Italy

Abstract

A relevant complexity factor in developing and testing mobile apps is given by their sensibility to changes in the context in which they run. As an example, apps running on a smartphone can be influenced by location changes, phone calls, device movements and many other typologies of context events.

In this paper, we address the problem of testing a mobile app as an event-driven system by taking into account both context events and GUI events. We present approaches based on the definition of reusable event patterns for the manual and automatic generation of test cases for mobile app testing.

One of the proposed testing techniques, based on a systematic and automatic exploration of the behaviour of an Android app, has been implemented and some preliminary case studies on real apps have been carried out in order to explore their effectiveness.

1. Introduction

Due to the relevant diffusion and success of mobile devices, there is a growing interest of the software engineering community for the world of mobile applications (mobile apps). According to Wasserman [23], there are several open issues regarding mobile app development and one of them regards the definition of effective strategies, techniques and tools for testing them.

Some specific characteristics of handheld devices must be carefully considered for mobile applications testing. They include heterogeneity of hardware configurations of mobile devices, scarceness of resources of the hardware platform, and variability of their running conditions.

Heterogeneity of mobile device platforms (that come equipped with diverse hardware sensors, screen displays, processors, etc.…) implies the need for expensive cross-platform development and testing [14]. The scarceness of resources of the hardware platform requires specific testing activities designed to reveal failures in the application behaviour due to resource availability (such as battery charge level, available RAM, wireless network bandwidth, …). The variability of running conditions of a mobile app depends on the possibility of using it in variable contexts, where a context represents the overall environment that the app is able to perceive. More precisely, Abowd et al. [1] define a context as: "any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is including the user and applications themselves".

When a mobile application has been designed to be aware of the computing context in which it runs and to adapt and react according to it, it belongs to the category of context-aware applications [6]. These apps may be notified of any change to their context by means of events.

Context awareness of mobile apps yields several new challenges for mobile app testing too, since an app should be tested in any environment and under any contextual input [16]. However, a considerable part of mobile app testing literature omits to consider the context-awareness issue [5], rather focuses on specific mobile problems such as testing in variable network conditions [18], security testing [9], performance testing [12], or GUI testing [2, 11]. Another part addresses context-aware testing issues [10, 13, 17, 21, 22] proposing solutions to the problem of context modeling and deriving test cases based on the proposed models.

In this paper, we focus on the problem of testing a mobile app taking into account its context and context-related events. To this aim, Section 2 presents possible strategies of event-based testing that take into account contextual events. Section 3 proposes how event-patterns could be used in three scenario-based mobile testing approaches. In Section 4 we present some technological solutions for implementing the proposed scenario-based testing techniques in the Android platform, while in Section 5 we show an example of using one of the proposed techniques for testing real Android apps. Section 6 presents the conclusions and future works.

2. Event-Based Testing Techniques for Mobile Apps

In event-based testing of event-driven systems (EDS), the behaviour of the system is checked with input consisting of specific event sequences [8].

Mobile apps are event-driven systems, too, but, differently from other traditional event-driven software systems, like GUIs or Web applications, they are able to sense and react to a wide set of events besides user ones. Current mobile devices are equipped indeed with a wide variety of hardware sensors that are able to sense the *context* in which the device stays and to notify context changes to the running app by means of events.

Therefore, since the user can be considered as a part of the context of an app [1], in event-based testing the application behaviour will have to be checked in response to several types of context event, such as:

- user events produced through the GUI;
- events coming from the external environment and sensed by device sensors (such as temperature, pressure, GPS, geomagnetic field sensor, etc.);
- events generated by the device hardware platform (such as battery and other external peripheral port, like USB, headphone, network receiver/sender, etc.);
- events typical of mobile phones (such as the arrival of a phone call or a SMS message);
- events like the arrival of an e-mail or social networks notifications, that are related to the fact that modern mobile phones are more and more Internet connected.

It is important to remark that not all the mobile apps are designed to react to context-related events not coming from the GUI. As an example, Muccini et al. [16] distinguish between apps, named MobileApps, that react to all contextual events (both GUI and non-GUI ones) to generate context-based outputs, and apps (called App4Mobile) that react only to GUI events, since they are just traditional applications that have been rewritten to run on mobile devices.

The category of App4Mobile may be effectively tested by testing techniques applicable to traditional applications (like GUI based testing for desktop or Web applications [24]). Vice-versa, MobileApps requires to be tested more carefully, by event-based testing techniques that properly consider all types of context-related events. This testing activity may be very expensive due to the large number of possible contexts, event classes and combinations of events and contexts to be considered.

Effective strategies for test case generation should be used to define the sequences of events of mixed types. To this aim, both "simple" strategies not requiring any specific knowledge about the app under test, and more "systematic" approaches based on such knowledge are usable. As an example, simple approaches may define event sequences just trying to achieve the coverage of each class of contextual events with a fair policy. This technique may help to discover unacceptable behaviours of the app (like crashes or freezes) that are often reported in bug reports of mobile apps and appear when the app is impulsively solicited by contextual events like the ones notifying connection/disconnection of a plug (USB, headphone, …), an incoming phone call, the GPS signal loss (for instance when the device enters a tunnel), and similar ones.

Other systematic test generation strategies may require the coverage of specific event sequences representing specific *usage scenarios of the application*.

Scenarios are software specifications defining relevant ways of exercising an application by sequences of events. They may be described by different formalisms, like MSC-Message Sequence Charts, State-Transitions systems, Event-Flow-Graphs, UML sequence diagrams, etc. There are several scenario-based testing approaches presented in the literature. As an example, in [15] scenarios are derived from Event-B model of the system under test (where an Event-B is a formalism for modelling the behaviour of event-based systems as a state transition system) and transformed into executable JUnit test cases thanks to Java language implementation templates. Garzon et al. [10] have presented an approach based on the DEVS formalism (a discrete event system specification) to model scenario-specific event-patterns. Starting from the model, it is possible to produce automatically several event sequences that represent valid sensor-related traces of a given scenario by means of an event generator application. This model-based approach facilitates and speeds up the generation of sensor-based data for context-aware applications testing. Wang et al. [22] present a technique for detecting faults in Context-Aware Adaptive Applications (CAAAs) by defining a formal, finite-state model of adaptation (A-FSM) and then analysing the model for finding adaptation faults. The A-FSM model represents the execution of a CAAA by explicitly connecting context updates with adaptations of the application and helps to isolate adaptation faults caused by erroneous rule predicates and asynchronous context updates.

In scenario-based testing, suitable techniques for obtaining scenarios are needed. An interesting approach may be based on "event-patterns", a representation of peculiar event sequences that abstract meaningful test scenarios. An event-pattern may involve one or more contextual entities and possibly trigger a faulty behaviour of the application. These patterns may be defined manually or on the basis of bug report analysis. Event-patterns are often used for rapid testing of embedded systems, too [20].

In the following section, we analyse possible approaches for using event-patterns for testing mobile applications.

3. Techniques for Event-Patterns Based Testing

An event-pattern can be defined as a notable sequence of contextual events that may be used to exercise the application. It may be specified by a name, a textual description and the corresponding event sequence that must include one or more events. The sequence can be defined by appropriate regular expressions specifying optional, mandatory, iterative events, etc.

As an example, Table 1 lists some event-patterns that have been manually defined after a preliminary analysis conducted on the bug reports of open source applications available at https://github.com and http://code.google.com.

An event-pattern may be included in other event sequences, or used in isolation to test an app.

Table 1. Some Event-Pattern Examples ("+" means one or more, "^" means not)

Event-Pattern Name	Event-Pattern Description	Event-Pattern Specification
LRGPS	Loss and successive Recovery of GPS signal while walking	(locationChange)+, GPSLoss, GPSRecovered, (locationChange)+
NI	Network Instability	(NetworkEnabled,NetworkDisabled)+
EGSW	The user Enables the GPS provider through the Settings menu and starts a Walk	openSettings, GPSOn, (locationChange)+
SIE	Arrival of a phone call when the device is in Stand-by. Then the call ends before the user accepts it.	standBy, IncomingPhoneCall, phoneCallEnds
UP	USB plugging in after any other event except the event itself	^USBPlugged,USBPlugged
MSVC	Magnetic Sensor value changed after any other event except the event itself.	^magneticSensor ValueChange, magneticSensor ValueChange
IPC	Incoming of a phone call after any other event except the event itself.	^IncomingPhoneCall, IncomingPhoneCall

For automating test execution, each event-pattern can be associated with a test class that exposes an `execute` method able to trigger the pattern's sequence of events.

As an example, Table 2 reports the `execute` methods of three patterns implemented in Java.

Once an event-pattern repository is available, it will be possible to generate test cases either manually or by semi-automatic approaches. In the following, we show three examples of testing techniques that exploit the event-patterns for testing a mobile app.

Manual technique (T1): A tester manually uses event-patterns to define scenario-based test cases that include one or more instances of event-patterns. The tester can add the needed assertions manually, or test cases can just check the occurrence of crashes.

As an example, let suppose that the tester wants to test the app behaviour in the following scenario:

The user activates the GPS provider in the settings menu of the application, and begins to cross the path that goes from point A to point B through N points. At point X of the navigation, the application loses the GPS signal and recovers it at the point Y.

This scenario includes instances of two event-patterns, EGSW and LRGPS respectively. The tester can reuse the code of these patterns to write the corresponding scenario test. Figure 1 shows the code implementing the scenario test reusing the pattern code.

Mutation-based technique (T2): Event-patterns are used to modify existing test cases, by applying mutation techniques that add event-pattern sequences inside already existing test cases (defined either manually or by Capture & Replay techniques or automatically).

As an example, the tester may want to prove the absence of crashes in an application scenario where, after any user event, the device goes in stand-by and a phone call comes (pattern SIE).

Another approach that could be used for mutating the test cases is the one proposed by Barbosa et al [7] for GUI testing. Test cases made by a sequence of events are altered automatically by introducing mutations in order to generate behaviours corresponding to errors users typically make.

```
public void testScenario00038() {
        EGSW.execute(A,X-1,routeAX);
        LRGPS.execute(X,Y,B,routeYB);
}
```

Figure 1: Scenario Test Case

Table 2. Examples of Implementations of Event-Patterns

Pattern	Pattern Class Signature	`Execute` Public Method

EGSW	`public Class EGWS {`
	`...`
	`  public void execute(Point A, Point B, ArrayList<Point> Route);`
	`  private void OpenSettings();`
	`  private void EnableGPS();`
	`  private void Navigate(Point A, Point B, ArrayList<Point> Route);`
	`...`
	`}`
		`public void execute (Point A, Point B, ArrayList <Point> Route){`
		`   OpenSettings();`
		`   EnableGPS();`
		`   Navigate(A,B,route);`
		`}`

LRGPS	`public Class LRGPS {`
	`...`
	`  public void execute(Point A, Point Y, Point B, ArrayList<Point> Route);`
	`  private void GPSLocationChange(Point X);`
	`  private void GPSLoss();`
	`  private void GPSRRecovered();`
	`  private void Navigate(Point A, Point B, ArrayList<Point> Route);`
	`...`
	`}`
		`public void execute (Point X, Point Y, Point B, ArrayList<Point> Route) {`
		`   GPSLocationChange(X);`
		`   GPSLoss();`
		`   GPSRRecovered();`
		`   GPSLocationChange(Y);`
		`   Navigate(Y+1,B,Route);`
		`}`

SIE	`public Class SIE {`
	`...`
	`  public void execute();`
	`  private void deviceGoesInStandBy();`
	`  private void incomingPhoneCall();`
	`  private void phoneCallEnds();`
	`...`
	`}`
		`public void execute() {`
		`   deviceGoesInStandBy();`
		`   incomingPhoneCall();`
		`   phoneCallEnds();`
		`}`

In this case, the tester may automatically modify existing JUnit test cases, obtained using an Android GUI Ripper [2, 3, 5], by applying the event-pattern SIE, as it is shown in Table 3.

Table 3. An Example of Mutation of a Test Case by an Event-Pattern

Test Case before mutation	Mutated test case

`public void testTrace00004() {`
 	
`  fireEvent (16908315, 16, "OK", "button", "click");`
 	
`  fireEvent (2131099651, 6, "", "button", "click");`
 	
`  fireEvent (0, "", "null", "openMenu");`
`}`
	`public void testTrace00004_EP_SIE() {`
	
	`  fireEvent (16908315, 16, "OK", "button", "click");`
	
	`  SIE.execute();`
	
	`  fireEvent (2131099651, 6, "", "button", "click");`
	
	`  SIE.execute();`
	
	`  fireEvent (0, "", "null", "openMenu");`
	
	`  SIE.execute();`
	`}`

Exploration-based technique (T3): This third technique establishes that event-patterns are used in automatic black-box testing processes based on dynamic analysis of the mobile app. In this case, an app exploration technique like the one reported in Figure 2 may be used to define test cases and execute them at the same time.

```
StartApplication();
while (! termination Criterion) {
        ExtractCurrentSensingEvents();
        PlanTasks();
        RunNextTask();
}
```

Figure 2: An Automatic Exploration-Based Testing Technique

The exploration technique first launches the app execution at a given start context (`StartApplication`), then iteratively detects the current set of sensing events of the app (`ExtractCurrentSensingEvents`) i.e., events that the app can sense and react to, plans how to fire them according to a given task planning strategy (`PlanTasks`), and then runs the next task (`RunNextTask`).

A task is a sequence of context events fired starting from the same initial start context of the app. The task planning strategy defines tasks by mixings context events either in systematically or in random manner [13]. As an example, a systematic planning strategy may establish that instances of UP and MSVC event-patterns are systematically executed after the execution of any other sensing event.

Although the proposed techniques are applicable to any mobile app, we have preliminary assessed their feasibility in the context of Android mobile applications. In particular, to implement the proposed techniques we had to solve two main technological problems related to the Android platform, that are:

a) defining a solution for dynamically recognizing the context event classes which the app is able to sense and react at a given time (i.e., implementing the operation `ExtractCurrentSensingEvents` in T3);
b) defining techniques for triggering the context events (i.e., implementing the `RunNextTask` operation in T3).

The proposed solutions are presented in the following section.

4. Implementing Event-Based Testing in the Android Platform

Android applications are event driven systems written in Java, where each app runs in its own process within its own instance of virtual machine. Each app is composed of several types of components instantiated at run-time (including Activities, Services, Broadcast Receivers, Content Providers). Among these components, Activity classes are responsible for managing the user interface of the device, with the constraint that just one instance at a time will be in the running state and will have the complete and exclusive control of the GUI, Service classes are able to execute computations in background, and Broadcast Receiver classes have the responsibility for the activation of proper components in response to specific events. Only these three types of components can be directly activated as a response to an event.

Run-time events can be handled either directly by the running component (if it defined an appropriate Listener to that event), or by the Android run-time environment using the mechanism of Intent Messages. Intent Messages are asynchronous messages that can be triggered either by the Android run-time environment in consequence of the occurrence of an event, or directly by the running component. The Android run-time environment realizes a late binding by dynamically searching for components that declared their ability in responding to a specific type of Intent Messages. These declarations are contained in the Android Manifest xml file; each application has its own unique manifest.

To solve the problem of dynamic recognition of the context event classes which the app is able to sense and react, we adopted two different solutions. Indeed, the set of context events that the app is able to sense and react to includes two distinct subsets. The former subset includes events that can be sensed by listeners and managed by the relative handlers defined by the running component itself. This set can be deduced by Java reflection techniques, since Android apps usually dynamically declare listeners at run-time and code static analysis would not suffice. The latter subset includes events that may be managed by other app components and notified by means of Intent Messages. This set can be obtained by means of static analysis of the Android Manifest xml file of the application by searching for intent-filter tags reporting the set of Intent Messages to which any component of the application is sensible.

For triggering the context events, we developed different techniques.

A first solution exploits the APIs provided by the `java.lang.reflection` package, a set of classes designed for dynamic querying of Java class instances. Using these APIs we are able to directly access and execute event handlers methods related to event listeners instantiated at a given time. This technique can be adopted to trigger any event having a registered event listener, such as GUI events.

As to the problem of raising sensor-related context events, we adopted a solution that fires fake events instead of real sensor events. To this aim, we substituted the Android Sensor Framework classes included in the android.hardware package (that is responsible for sensor management in Android) with an ad hoc modified version of this package that includes classes for generating fake events. This solution disables the sensibility of the app to real sensor events at all. A similar solution has been implemented in the OpenIntents SensorSimulator project [19], too.

As to the emulation of location changes of the device we exploited the APIs of the `LocationManager` class provided by Android. This class allows the system location services to be accessed. The services are used to obtain periodic updates of the device's geographical location, or to fire an application-specified Intent when the device enters in the proximity of a given geographical location. To emulate the device location change notification events we've exploited the `addTestProvider()` method that creates a mock location provider that programmatically emulates location changes.

The techniques presented above can be used to implement testing tools supporting the execution of the techniques shown in section 4. In particular, we have used them to develop a tool that implements the T3 technique.

5. A Case Study

We conducted an exploratory case study with the purpose of assessing how the effectiveness of an event-based testing technique varies when context events, not only GUI events, are taken into account. In particular, we decided to analyse an automatic testing technique, like the proposed Exploration-based technique T3.

For this aim, we considered some real-world Android applications and each of them was tested by the T3 technique twice. The first time, the planning strategy was configured to perform systematically simple patterns that exercise only user events. The second time, we choose to execute simple patterns each one including a different type of context event. Lastly, we compared the testing effectiveness, in terms of code coverage, achieved by each of the two executions.

To perform this experiment we exploited our Android Ripper tool [3]. In the first execution of the technique, the Android Ripper was configured to trigger only user events [2], while the second time we exploited another version of the Ripper, called Extended Ripper, that was able to fire context events such as location changes, enabling/disabling of GPS, changes in orientation, acceleration changes, reception of SMS messages and phone calls, shooting of photos with the camera. Both versions of the Ripper are able to systematically explore the app under test by searching for crashes, to measure the obtained code coverage and to automatically generate Android JUnit test cases reproducing the explored executions.

In the experiment we tested five real Android applications. They belonged to the category of Mobile Apps and used context data of different types. They are all published in the Android Market (https://play.google.com/store) and their source code is freely available. Table 4 reports a brief description of their features.

Each app was exercised from the same starting context both by the Android Ripper and by the Extended Ripper. We measured the resulting code coverage in terms of lines of code (LOCs) and methods: Table 5 shows the coverage values we obtained.

Table 4. Characteristics of the Tested Apps

App	Description
Marine Compass [https://play.google.com/store/apps/details?id=net.pierrox.mcompass]	App showing a virtual marine compass providing the correct north direction on the basis of the data provided by the orientation sensor
Bubble Level [https://play.google.com/store/apps/details?id=net.androgames.level]	App transforming the device in a virtual spirit level on the basis of the data obtained by the orientation sensor and by the accelerometer
Pedometer [https://play.google.com/store/apps/details?id=name.bagi.levente.pedometer]	App showing a set of statistics regarding the walking of a person on the basis of accelerometer data
Omnidroid [https://play.google.com/store/apps/details?id=edu.nyu.cs.omnidroid.app]	App for managing of personal actions and tasks that takes into account received/sent SMS messages and phone calls.
Wordpress for Android [https://play.google.com/store/apps/details?id=org.wordpress.android]	Client for managing Wordpress blogs, that can interact with the camera for the insertion of photos in a blog and with the GPS for the insertion of localization data

Table 5. Code Coverage Results

App	LOC Coverage		Method Coverage	
	Android Ripper	Extended Ripper	Android Ripper	Extended Ripper
Marine Compass	435 (92%)	463 (97%)	25 (86%)	27 (93%)
Bubble Level	371 (60%)	464 (75%)	75 (65%)	85 (74%)
Pedometer	528 (66%)	544 (67%)	160 (71%)	161 (72%)
Omnidroid	3409 (56%)	3480 (57%)	789 (58%)	813 (60%)
Wordpress	4505 (45%)	4599 (46%)	779 (53%)	784 (53%)

As the data show, the LOC code coverage of the first two Mobile Apps, Marine Compass and Bubble Level, achieved by the Extended Ripper grew of about 5% for Marine Compass and 15% for Bubble Level with respect to the Ripper one. Analogously the method coverage increased of 7% and 9%, respectively.

This difference could be attributed to the fact that a relevant part of app code implementing context event handling was covered just by the Extended Ripper.

For the Pedometer app, the Extended Ripper reached just a slight increase in coverage with respect to the GUI Ripper (about 1% additional LOC and method coverage). This datum depended on the presence of just one context-related event handler in the app code, the one responsible for managing the acceleration change notification event.

As to the last two apps, Omnidroid and Wordpress for Android, the Extended Ripper coverage slightly increased with respect to the Ripper coverage (just 1% more LOC coverage). In particular, in Omnidroid the Extended Ripper covered also the event handlers related to the management of incoming phone calls and SMS messages, while in Wordpress it covered the code of the camera photo shoot notification event handler, as well as the location change notification one.

These results show that event-based testing effectiveness actually improves thanks to the considered comprehensive set of context events. The more the app uses data from the context, the more the improvement becomes relevant. This datum preliminarily showed the utility of the proposed techniques.

6. Conclusions and Future Works

In this paper, three proposals of techniques for event based testing of mobile applications have been presented. These techniques take into account both user events and context events.

The proposed testing techniques involve the manual definition of reusable event patterns including context events.Event patterns may be used to manually generate test cases, to mutate existing test cases, and to support the systematic exploration of the behaviour of an app. This last technique represents an extension of the GUI Ripping technique that we have presented in the past [2] and that systematically explores the behaviour of an application by triggering only user events. We presented the application of this exploration technique to Android apps, by addressing the technological problems related to the search of fireable context events and to their emulation. Some preliminary case studies conducted on real world Android apps demonstrate the effectiveness of the implemented technique in terms of code coverage.

Some future works can be addressed on the basis of the ideas presented in this work. As regards event-patterns, in future we want to build an event-patterns repository by analysing a large corpus of bug reports related to mobile apps and to implement tools supporting the automatic injection of event patterns in existing test cases.

As regards technological aspects of mobile Android applications, we want to consider events causing the interaction between different components of the same application or different applications, by adding to the Extended Ripper new features supporting Intent Messages generation and execution.

Finally, wider experimentation will be addressed in order to assess the effectiveness of the proposed techniques also in terms of fault-detection.

7. Acknowledgments

This work was partially supported by grants from the Italian ministery for university and research (MIUR) realted to the project IESWECAN (Informatics for Embedded SoftWare Engineering of Construction and Agricultural machiNes - PON01-01516).

References

[1] G. D. Abowd, A. K. Dey, P. J. Brown, N. Davies, M. Smith and P. Steggles. Towards a Better Understanding of Context and Context-Awareness. In Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing (HUC '99), Springer-Verlag, 1999, pp. 304-307.

[2] D. Amalfitano, A.R. Fasolino, P.Tramontana, S. De Carmine, and A. M. Memon. Using GUI ripping for automated testing of Android applications. Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012), 2012. ACM, pp. 258-261.

[3] D. Amalfitano, A.R. Fasolino, P. Tramontana, S. De Carmine, G. Imparato. A Toolset for GUI testing of Android Applications. Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM), 2012, IEEE CS Press, pp.650-653.

[4] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. Robbins. Testing Android Mobile Applications: Challenges, Strategies, and Approaches, Advances in Computers, Volume 89, 2013, Elsevier, pp. 1–52.

[5] D. Amalfitano, A.R. Fasolino, P. Tramontana. A GUI Crawling-Based Technique for Android Mobile Application Testing. IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2011, IEEE CS Press, pp. 252,261.

[6] M. Baldauf, S. Dustdar and F. Rosenberg. A survey on context aware systems. Int. J. Ad Hoc Ubiquitous Comput. 2, 4 (June 2007), Inderscience Publishers, pp.263-277.

[7] A. Barbosa, A.C.R. Paiva, J. Creissac Campos. Test case generation from mutated task models. Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems (EICS '11). ACM, New York, NY, USA, pp. 175-184.

[8] F. Belli, M. Beyazit, A. Memon. Testing is an Event-Centric Activity. Proceedings of the IEEE Sixth International Conference on Software Security and Reliability Companion (SERE-C). 2012, IEEE CS Press, pp.198-206.

[9] W. Enck, M. Ongtang, P. McDaniel. Understanding Android Security. Security & Privacy, IEEE, vol.7, no.1, Jan.-Feb. 2009, pp. 50-57,

[10] S. Rodriguez Garzon and D. Hritsevskyy. Model-based generation of scenario-specific event sequences for the simulation of recurrent user behavior within context-aware applications. Proceedings of the 2012 Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium (TMS/DEVS '12). Society for Computer Simulation International, Article 29, 6 pages.

[11] C. Hu and I. Neamtiu. Automating GUI Testing for Android Applications. Proc. of AST 2011, 6th international workshop on Automation of software test, ACM Press, pp. 77- 83.

[12] H. Kim, B. Choi, W. Eric Wong. Performance Testing of Mobile Applications at the Unit Test Level. Proc. of 2009 Third IEEE International Conference on Secure Software Integration and Reliability Improvement, IEEE Comp. Soc. Press, pp. 171- 181.

[13] Z. Liu, X. Gao, and Xiang Long. Adaptive Random Testing of Mobile Application. 2nd International Conference on Computer Engineering and Technology (ICCET), 2010, vol.2, pp 297-301.

[14] A. K. Maji, K. Hao, S. Sultana, S. Bagchi, Characterizing Failures in Mobile OSes: A Case Study with Android and Symbian. Proceedings of the 21st IEEE Int. Symposium on Software Reliability Engineering, 2010, IEEE CS Press, pp. 249-258.

[15] Q.A. Malik, J. Lilius, L. Laibinis. Scenario-Based Test Case Generation Using Event-B Models. Proceedings of the First International Conference on Advances in System Testing and Validation Lifecycle, 2009. IEEE CS Press, pp. 31-37.

[16] H. Muccini, A. Di Francesco, P. Esposito. Software Testing of Mobile Applications: Challenges and Future Research Directions. Proceedings of the 7th International Workshop on Automation of Software Test (AST), 2012,

[17] M. Sama, D. S. Rosenblum, Z. Wang and S. Elbaum. Model-based fault detection in context-aware adaptive applications. Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering (SIGSOFT '08/FSE-16). 2008. ACM, pp. 261-271.

[18] I. Satoh. Software testing for wireless mobile application. IEEE Wireless Communications, Oct. 2004, IEEE CS Press pp. 58-64.

[19] OpenIntents Sensor Simulator, http://code.google.com/p/openintents/wiki/SensorSimulator

[20] W.Tsay, L Yu, F. Zhu, R. Paul. Rapid Embedded System Testing Using Verification Patterns. IEEE Software, 2005, Vol. 22, 4, IEEE CS Press, pp.68-75.

[21] T. Tse, S. Yau, W. Chan, H. Lu, and T. Chen. Testing context-sensitive middleware-based software applications. Proceedings of International Computer Software and Applications Conference, 2004, IEEE CS Press, pp. 458–465.

[22] Z. Wang, S. Elbaum, D.S. Rosenblum. Automated Generation of Context-Aware Tests. Proceedings of the 29th International Conference on Software Engineering, ICSE 2007. IEEE CS Press, pp.406-415.

[23] A. I. Wasserman. Software engineering issues for mobile application development. Proceedings of the FSE/SDP workshop on Future of software engineering research (FoSER '10), 2010, ACM, pp. 397-400.

[24] X. Yuan, M.B. Cohen, A.M. Memon. GUI Interaction Testing: Incorporating Event Context. IEEE Transactions on Software Engineering, vol.37, no.4, July-Aug. 2011. IEEE CS Press, pp.559-574.