A GUI Crawling-based technique for Android Mobile Application Testing

Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana

domenico.amalfitano@unina.it, anna.fasolino@unina.it, porfirio.tramontana@unina.it

Dipartimento di Informatica e Sistemistica, Università di Napoli Federico II,
Via Claudio 21, 80125 Napoli, Italy

Abstract

As mobile applications become more complex, specific development tools and frameworks as well as cost-effective testing techniques and tools will be essential to assure the development of secure, high-quality mobile applications.

This paper addresses the problem of automatic testing of mobile applications developed for the Google Android platform, and presents a technique for rapid crash testing and regression testing of Android applications. The technique is based on a crawler that automatically builds a model of the application GUI and obtains test cases that can be automatically executed. The technique is supported by a tool for both crawling the application and generating the test cases. In the paper we present an example of using the technique and the tool for testing a real small size Android application that preliminary shows the effectiveness and usability of the proposed testing approach.

1. Introduction

With about three billion people using mobile phones worldwide and the number of devices that can access the net climbing rapidly, the future of the Web is definitely mobile.

Bridging the gap between desktop computers and hand-held devices is the main challenge that research in mobile applications is addressing for the next future: according to Andy Rubin, Guru for Google's Android, “There should be nothing that users can access on their desktop that they can’t access on their cell phone”.

Thanks to the advancement of hardware industry, modern mobile phones have now faster processors, growing memories, faster Internet connections, and much richer sensors, and are able to host more demanding applications. Moreover, the current applications programming platforms and development tools used to develop applications for mobile devices (such as Java ME, .NET Compact Framework, Flash Lite, Android) provide options to create highly functional mobile multimedia applications [7], allowing the use of various technologies, like Java, Open C, Objective C, Python, Flash Lite or Web technologies.

In such a scenario, the complexity, variety and functional richness of mobile applications are growing and the request for mobile software applications offering even more complex, rich, and usable functionalities is going to grow more and more in the next future.

Unfortunately, the quality of applications for mobile devices is often poor. This lack of quality is mostly due to very fast development processes where the testing activity is neglected or carried out in a superficial way since it is considered too complex, difficult to automate, expensive and time-consuming. Indeed, testing a mobile device application is not a trivial task due to several factors: a first factor consists of the variety of input that normally solicit a mobile application (such as user input, context and environment inputs) which makes it hard to find the right test cases that expose faults. A second factor is the heterogeneity of the technologies used by the devices, so that multiple tests on multiple platforms should be performed.

In order to obtain higher quality mobile applications, greater attention should be devoted to the testing activity throughout the development process and effective models, methods, techniques and tools for testing should be available for testers. In particular, cost-effective, rapid, and automated testing processes should be executed when possible, in order to cope with the fundamental necessity of the rapid delivery of these applications.

This paper focuses on the problem of automatic testing of mobile applications developed for the Google Android platform. Among the currently available mobile platforms (such as Symbian, Android, Research In Motion and Apple iOS), Android is predicted to become the second largest mobile Operating System by 2012 [6], thanks to the open-source nature and the programmability features: Android is indeed based on open source Linux software that allows developers to access to the underlying code. This feature will certainly increase Android diffusion in the market of mobile devices.

Android applications can be actually considered Event Driven Software (EDS) whose behaviour is driven by several types of events. Hence, a major issue in Android application testing is that of assessing which testing approaches usable for traditional EDS systems (such as GUIs, Rich Internet Applications, embedded software, etc.) are also applicable for Android based mobile applications and which tuning and technological adaptations are needed for them.

In particular, in the paper we focus on GUI testing techniques already adopted for traditional applications and propose a GUI crawling based technique for crash testing and regression testing of Android applications. The technique is supported by a tool for producing test cases that can be automatically executed.

The paper is organized as it follows. Section 2 describes the main features of an Android application and the principal open issues concerning Android application testing. Section 3 presents the proposed testing technique while Section 4 provides a description of the supporting tool. An example of using the technique is illustrated in Section 5 and Section 6 discusses related works. Eventually, Section 7 provides conclusions and future works.

2. Background

The Android Developers Web site [2] defines Android as a software stack for mobile devices that includes a Linux-based operating system, middleware and core applications. Using the tools and the APIs provided by the Android SDK, programmers can access the stack resources and develop their own applications on the Android platform using the Java programming language. Although based on well-known open source technologies like Linux and Java, Android applications own remarkable peculiar features that must be correctly taken into account when developing and testing them. In the following, we present an insight into Android application internals and focus on the technological approaches adopted for developing user interfaces and event handling in user oriented applications.

2.1 Implementing the GUI of an Android Application

The Android operating system is often installed on smartphone devices that may have limited hardware resources (like CPU or memory) and a small-sized screen, but are usually equipped with a large number of sensors and communication devices such as a microphone, wi-fi and Bluetooth chips, GPS receiver, single or multi touch screen, inclination sensors, camera and so on. In order to optimize the management of all these resources and to cope with the intrinsic hardware limitations, the Android applications implement a multi-thread process model in which only a single thread can access to user interface resources, while other threads contemporarily run in background. Moreover, each application runs in its own virtual machine (the Dalvik one) that is a virtual machine optimized for Android mobile devices.

An Android application is composed of several types of Java components instantiated at run-time (namely, Activities, Services, Broadcast Receivers, and Content Providers) where the Activity components are crucial for developing the user interface of an application [2]. The Activity component, indeed, is responsible for presenting a visual user interface for each focused task the user can undertake. An application usually includes one or several Activity classes that extend the base Activity class provided by the Android development framework. The user interface shown by each activity on the screen is built using other framework classes such as View, ViewGroup, Widget, Menu, Dialog, etc.

In its lifecycle, an Activity instance passes through three main states, namely *running*, *paused* and *stopped*. At run-time just one activity instance at the time will in the *running* state and will have the complete and exclusive control of the screen of the device. An Activity instance can make dynamic calls to other activity instances, and this causes the calling activity to pass to the *paused* state. When a running activity becomes paused then it has lost focus but is still visible to the user. Moreover, an activity can enter the *stopped* state when it becomes completely obscured by another activity.

In Android applications, processing is event-driven and there are two types of events that can be fired (e.g., user events, and events due to external input sources). The user events (such as Click, MouseOver, etc.) that can be fired on the user interface items (like Buttons, Menu, etc.) are handled by handlers whose definition belong either to the respective interface object, or to the related Activity class instance (using the Event Delegation design pattern). As to the events that are triggered by other input sources, such as GPS receiver, phone, network, etc., their handling is always delegated to an Activity class instance.

2.2 Open Issues with Android Application Testing

Since the behaviour of an Android application is actually event-driven, most of the approaches already available for EDS testing are still applicable to Android applications. However, it is necessary to assess how these techniques can be adopted to carry out cost-effective testing processes in the Android platform.

Most of the EDS testing techniques described in the literature are based on suitable models of the system or sub-system to be tested like Event-Flow Graphs, Event-Interaction-Graphs, or Finite State Machines [4, 11, 13], exploit the analysis of user session traces for deriving test cases [1], or are based on GUI rippers [12] or Web application crawlers [15] that automatically deduce possible sequences of events that can be translated into test cases.

Using such techniques for the aims of Android testing will firstly require an adaptation of the considered models and strategies in order to take into account the peculiar types of event and input source that are typical of Android devices.

As a consequence, new reverse engineering and GUI ripping techniques will have to be designed for obtaining the necessary models, as well as platforms and tools aiding user session analysis, will have to be developed.

From the point of view of the supporting technologies, the Android development environment [2] provides an integrated testing framework based on JUnit [8] to test the applications. At the moment, the framework has been mostly proposed to carry out assertion based unit testing and random testing of activities. A further issue consists of assessing what support it is able to offer to the implementation of other automatic testing techniques too.

3. A Technique for Testing Android Applications

Like the crawler-based technique presented by [15] for testing Ajax applications, the automatic testing technique we propose for Android applications is based on a crawler that simulates real user events on the user interface and infers a GUI model automatically. The GUI model is hence used for deriving test cases that can be automatically executed for different aims, such as crash testing and regression testing.

The model produced by the crawler is actually a GUI Tree, the nodes of which represent the user interfaces of the Android application, while edges describe event-based transitions between them.

For obtaining this model, while the crawler fires events on the application user interface, it also captures data about interfaces and events that will be also used to decide the further events to be fired.

The data analysed by the crawler at run time belong to the conceptual model of an Android GUI that is represented by the class diagram shown in Figure 1.

The model shows that a GUI is made up of *interfaces* linked to each other by a *Transition* relationship. Each interface is characterized by the *Activity instance* that is responsible for drawing it and is composed by a set of Widgets. We define a *Widget* as a visual item of the Interface. A Widget can be implemented in the Android framework by an instance of a View class, a Dialog class or a Menu Item class.

Any Widget is characterized by a set of *Properties* with related *Values* (such as size, color, position, caption and so on). Some Widget Properties are *Editable*: in this case their values are provided as user input at run time (as an example, we can consider the text field of a TextView object).

Events can cause transitions between Interfaces. In Android applications there can be both user events and events associated with interrupt messages sent from any component making up the device equipment (such as GPS, phone, wireless connections, inclination sensors, etc.).

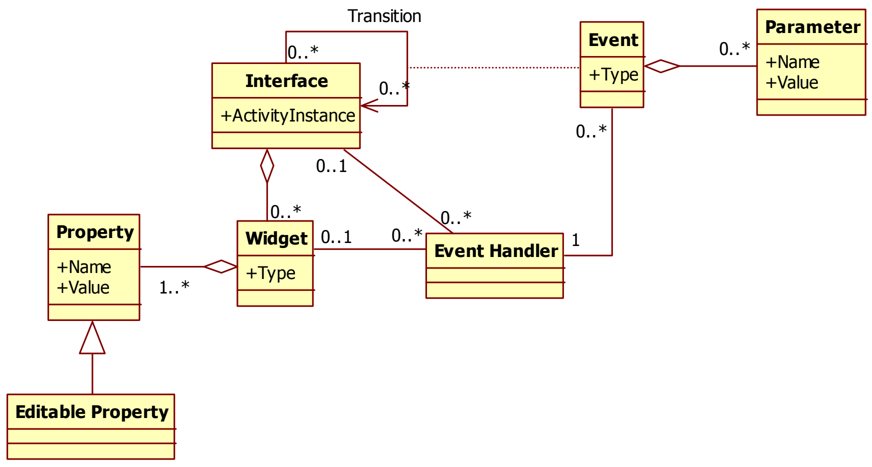

Fig. 1: Conceptual Model of an Android Application GUI

Event Handlers code can be either defined in the context of a Widget of the interface, or in the context of an Activity, depending on the type of Event. Events may have zero or more *Parameters* and each Parameter has a Name and a Value.

The GUI crawler builds the GUI tree using an iterative algorithm that relies on two main temporary lists (*Event* list and *Interface* list, respectively) and executes the steps reported in Figure 2.

0) Describe the starting interface (associated with the first interface shown by the application at its launch) in terms of its activity instance, widgets, properties and event handlers, and store this description into the Interface list;

1) Detect all the interface fireable events having an explicitly defined Event Handler and, for each event, define a possible way of firing it by choosing the random values that will be set into the widget Editable Properties and to the Event Parameter Values (if they are present). Save this information into an Event description and store this description into the Event List[1].

repeat
2) Choose one fireable event *E* from the Event List, set the needed preconditions and fire it, according to its description.

3) Catch the current interface and add a node representing that interface to the GUI tree; then add an edge between the nodes associated with the consecutively visited interfaces.

4) Describe the current interface in terms of all its properties, store the interface description in the Interface List, and check whether the current interface is 'equivalent' to any previously visited one, or it is a 'new' one. If it is equivalent to any interface or it does not include fireable events, the corresponding GUI node will be a leaf of the tree, otherwise the new interface fireable events will be detected and a description of each event will be defined and added to the Event List. In both cases, the E Event that caused that interface to be reached will be removed from the Event List.

Until the fireable Event list is empty

Figure 2: The Crawling algorithm

A critical aspect of any GUI crawling algorithm consists of the criterion used for understanding when two interfaces are equivalent. Several approaches have been proposed in the literature to solve this problem [2, 11, 15]. Our algorithm assumes two interfaces to be equivalent if they have the same Activity Instance attribute (see the model in Figure 1) and they have the same set of Widgets, with the same Properties and the same Event Handlers.

Another critical aspect of this algorithm consists of the approach it uses for defining the values of widgets' properties and event parameters that must be set before firing a given event. At the moment, the crawler assigns them with random values.

3.2 Test Case Definition

The GUI tree generated by the crawler is the starting point for obtaining test cases that can be run both for automatic crash testing and for regression testing of the application.

According to Memon et al. [13], crash testing is a testing activity that aims at revealing application faults due to uncaught exceptions.

To detect crashes in the subject Android application, we have implemented a technique based on a preliminary instrumentation of the application code that automatically detects uncaught exceptions at run-time. In this way, during the GUI exploration performed by the crawler we are able to perform a first crash testing. Indeed, test cases used for crash testing are given by the sequences of events associated with GUI tree paths that link the root node to the leaves of the tree.

As to the regression testing activity that must be executed after changes to a given application have been made, it is usually performed by rerunning previously run tests and checking whether program behaviour has changed and whether new faults have emerged.

In the regression testing of an Android application, we propose to use the same test cases used for crash testing, and we had to define a suitable solution to check possible differences between the application behaviours.

A possible way of detecting differences is by comparing the sequences of user interfaces obtained in both the test runs. The interface comparison can be made using test oracles having different degrees of detail or granularity [14]. As an example, the Monkey Runner tool [17] executes regression testing of Android applications but it checks results just by comparing the output screenshots to a set of screenshots that are known to be correct.

We propose to check whether all the intermediate and final Interfaces obtained during test case rerunning coincide with the ones obtained in the previous test

[1] In the Event List, the description of each event will include the sequence of events that need to be fired before firing that event. This sequence actually represents the pre-conditions for firing the event.

execution, and their Activity, Event Handlers, and Widgets' Properties and Values are the same. To do this checking, we add specific assertions to the original test cases that will be verified when tests are run.

A test will reveal a failure if any assertion is not verified, or some event triggering is not applicable.

4. The Testing Tool

In this section a tool supporting the testing technique proposed in the previous section will be presented.

The tool, named A^2T^2 (Android Automatic Testing Tool), has been developed in Java and is composed of three main components: a Java code instrumentation component, the GUI Crawler and the Test Case Generator.

The Java code instrumentation component is responsible for instrumenting the Java code automatically, in order to allow Java crashes to be detected at run-time.

The GUI crawler component is responsible for executing the Android crawling process proposed in section 3. It produces a repository describing the obtained GUI Tree, comprehending the description of the found Interfaces and of the triggered Events. Moreover, it produces a report of the experienced crashes, with the event sequences producing them.

The GUI crawler exploits Robotium [18], a framework originally designed for supporting testing of Android applications. Robotium provides facilities for the analysis of the components of a running Android application.

The GUI crawler extracts information about the running Activity, the Event Handlers that the Activity implements and the Widgets that it contains (with related Properties, Values and Event Handlers). Moreover, the GUI crawler is able to emulate the triggering of Events and to intercept application crashes.

The current prototype of A^2T^2 manages only a subset of the possible Widgets of an Android application, comprehending, TextView labels, TextEdit fields, Buttons and Dialogs, while, in the future, we plan to extend the support to a larger number of Widgets and Event typologies.

The Test Case Generator component is responsible for the abstraction of executable test cases supporting crash testing and regression testing from the GUI Tree produced by the GUI Crawler component.

Test cases produced by the Test Case Generator are Java test methods that are able to replay event sequences, to verify the presence of crashes (for crash testing) and to verify assertions regarding the equivalence between the Interfaces obtained during the replay and the original ones obtained in the exploration process (for regression testing).

Generated Test Cases exploit the functionalities offered by the Robotium framework both for events triggering and for the extraction of information about the obtained Interfaces.

Both the Crawler and the Generated Test Cases can be executed in the context of the Android Emulator provided by the Android SDK [3].

5. An Example

In this section we show an example of using the proposed technique and tool for testing a simple Android application.

The subject application implements a simple mathematic calculator that can operate either in a basic mode, providing the possibility of executing the basic arithmetic operations between numeric input values, or in a scientific mode, providing trigonometric functions, inverse trigonometric functions and other ones.

The application was developed for the Android 2.2 platform by using the libraries provided by the corresponding SDK. It consists of five Java classes contained in one package, for a total of 557 Java LOCs. Two of the implemented classes extend the Android Activity class and contain, in total, 36 different Widgets, comprehending Buttons, EditText and TextView Widgets.

After a preliminary automatic instrumentation of the application - that was needed for detecting runtime crashes - the application crawling was automatically executed by the tool and a GUI tree of the application was obtained. During crawling, 19 Events were triggered, 19 Interfaces were obtained, and an exception causing an application crash occurred. Using the equivalence criterion presented in section 3.1, the 19 Interfaces we obtained were grouped into the following three equivalence classes:

- Class IC1 that comprehends the Interfaces I1, I2, I3, I4, I5, I9, I16 corresponding to instances of the BaseCalculator Activity, by means of which the basic arithmetic operations can be performed (an example of an Interface belonging to IC1 is reported in Figure 3-a);
- Class IC2, comprehending the Interfaces I6, I7, I8, I10, I11, I12 and I19, corresponding to instances of the ScientificCalculator Activity, by means of which the trigonometric functions, the reciprocal function and the square root function can be computed (an

example of Interface belonging to IC2 is reported in Figure 3-b);

- Class IC3, comprehending the Interfaces I13, I14, 15, I17 and I18, corresponding to instances of the ScientificCalculator Activity by means of which inverse trigonometric functions, the reciprocal function and the square function can be computed (an example of Interface belonging to IC3 is reported in Figure 3-c).

Figure 4 shows the GUI Tree we obtained, where each node reports the screenshot and the label associated to the corresponding interface, and edges are labeled by the event that caused the transition between the interfaces. The leaves of the tree correspond always to interfaces that were equivalent to at least another interface previously explored by the crawler (the number in the Interface label represents, too, the order in which the Interface was found by the crawler).

As an example, our crawling technique was able to distinguish automatically the instances of Interfaces belonging to IC1 from interfaces of the other groups because they were associated with instances of different Activity classes. Moreover, it was able to distinguish between instances of Interfaces belonging to IC2 and IC3, because they included different sets of Buttons.

While exploring the GUI interfaces via the crawler some crashes of the application were discovered, too. As an example, a crash occurred after firing the E18 Event that corresponds to the click on the '*atan*' Button on the Interface I13.

The cause of this crash was the lack of the try/catch code block for handling the exception due to the input of a non-numeric value in the Input TextEdit widget. This caused a java.lang.NumberFormatException when the application tries to convert the string in the input field into a Double value before computing the arctangent function. After correcting this defect, we run the crawler again and obtained a new GUI tree where another instance of Interface (belonging to IC3 group) was correctly associated with the right node.

After obtained the GUI Tree, the Test Case Generator produced 17 test cases for crash testing that corresponded to the 17 different paths from the root to the leaves of the tree. The Test Case Generator tool developed 17 test cases for regression testing, too.

In order to assess the effectiveness of our test cases for the aims of regression testing, we injected two faults in the Android application and run the 17 regression test cases to find these faults.

The first injected fault was due to a change of the code of the Scientific Calculator Activity causing an interface Button (namely the one that makes it possible to return to the Base Calculator) to be no more drawn on the screen window.

One of the regression test cases (namely the test case corresponding to the execution of the event sequence E5-E12-E13) revealed an assertion violation. The assertion violation was due to a layout difference between the obtained Interface I13 and the corresponding one collected during the previous crawling process, since the new Interface did not contain the Button that was included in the original one.

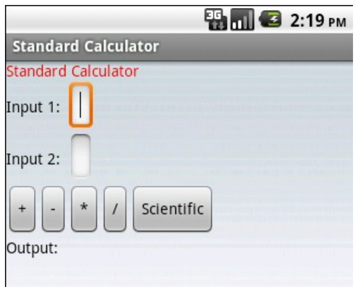

Figure 3-a: IC1 Interface

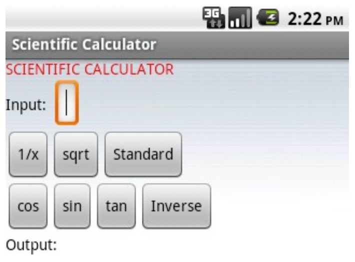

Figure 3-b: IC2 Interface

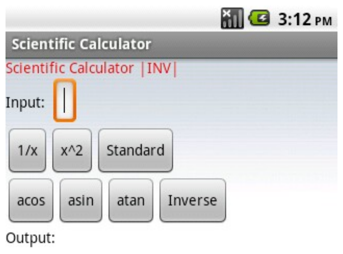

Figure 3-c: IC3 Interface

Figure 3: Screenshots of Interfaces of the example Android application

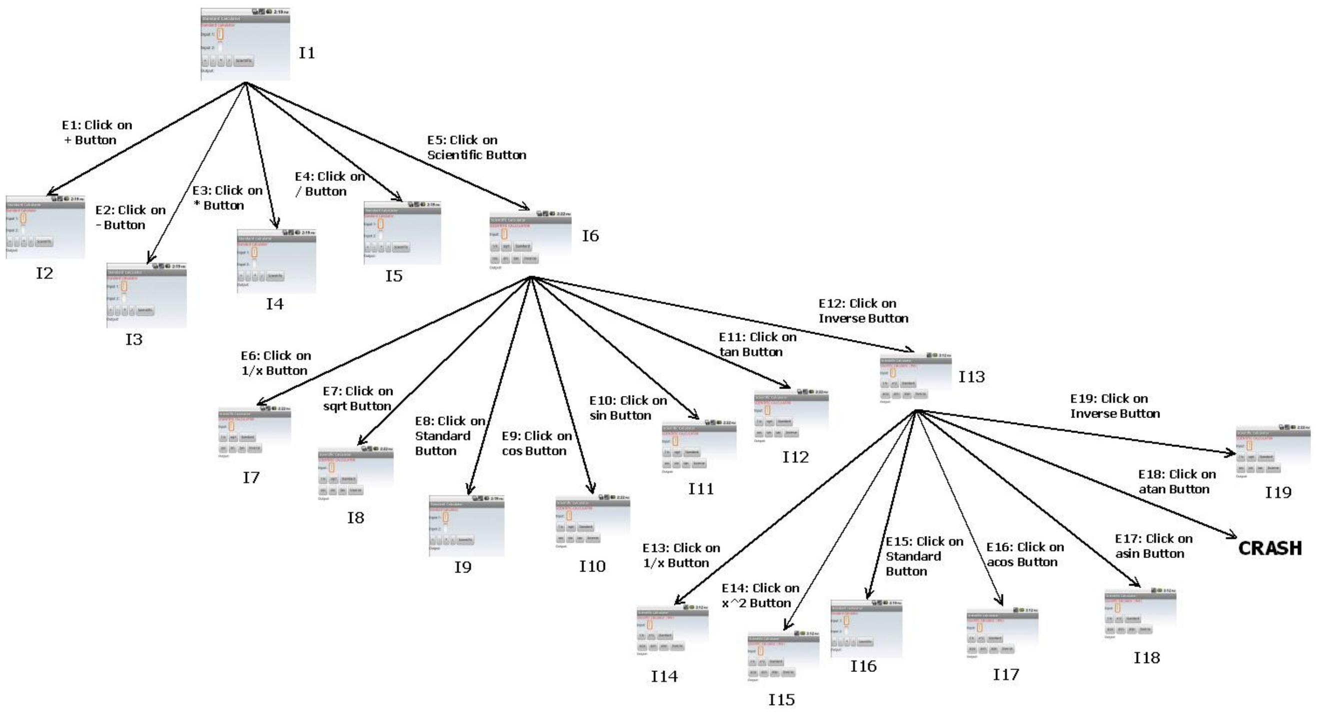

Figure 4: The GUI Tree obtained by crawling the example Android application

Figure 5 shows the Java code of the test case corresponding to the execution of the event sequence E5-E12-E13 that detected the fault.

```
public void testSequence11() throws Exception {
        InterfaceComparator.compare("I1");
        solo.clickOnButton("Scientific");
        InterfaceComparator.compare("I6");
        solo.clickOnButton("Inverse");
        InterfaceComparator.compare("I13");
        solo.enterText("Input", "dfghfdjg");
        solo.clickOnButton("1/x");
        InterfaceComparator.compare("I14");
}
```

Figure 5: Java code of the test case firing the E5-E12-E13 event sequence

In the Figure, '**`solo`**' is one of the classes that Robotium provides for automatically firing events onto the application, while '**`InterfaceComparator`**' is a class that we developed, having a method '**`compare`**' that is used to check the coincidence between interfaces.

The second fault we injected consisted of associating an incorrect event handler to the click event on the cosine Button (e.g., the '`calculateSin`' function) instead of the correct one (e.g., the '`calculateCos`' function). This fault is explained by the code fragment shown in Figure 6, where in the last line of code, `calculateCos` should be written instead of `calculateSin`.

```
View.OnClickListener calculateSin = new
View.OnClickListener() {
  public void onClick(View v) { … }
};

View.OnClickListener calculateCos = new
View.OnClickListener() {
  public void onClick(View v) { … }
};

sinButton.setOnClickListener(calculateSin);
cosButton.setOnClickListener(calculateSin);
```

Figure 6: Code fragment associated with an injected fault

The execution of the test case corresponding to the event sequence E5-E10 revealed an assertion violation and allowed the injected fault to be discovered. The violation was due to the difference between the obtained Interface and the one collected during the crawling process, since they contained different methods associated to the onClickListener attribute of cosButton widget.

We explicitly remark that, thanks to the type of assertion checked by our regression test cases, we were able to find a fault whose effects were not visible on the GUI. Other regression testing tools like Monkey Runner could not discover it, since it just limits itself to check screenshots.

However, the fault detection effectiveness of the technique depends considerably on the strategy used by the crawler for defining the input values needed for firing the events. As an example, a possible fault in the reciprocal function due to an unmanaged exception of a division by zero might be revealed only by a test case with a zero value in the input field. This value may not be used in any test case, due to the random strategy used by the crawler for generating input. Other input generation techniques should be considered in order to solve this problem.

Moreover, in the example we assumed that the replay of the same event sequence with the same input values produced always the same effects. In general, instead, the problems related to the management of preconditions and postconditions related to persistent data sources (such as files, databases, Shared Preferences objects, remote data sources) must be considered, too.

In conclusion, this example showed the usability of the technique for running crash testing and regression testing, and its effectiveness in detecting some types of fault in a completely automatic manner.

6. Related Works

As mobile applications become more complex, specific development tools and frameworks as well as software engineering processes will be essential to assure the development of secure, high-quality mobile applications. According to Wasserman [22], there are important areas for mobile software engineering research, and defining testing methods for product families, such as Android devices, is one of the areas requiring further efforts and investigations.

In the literature, recent works in testing mobile applications have mostly focused on the definition of frameworks, environments and tools supporting testing processes in specific development contexts. Other works have addressed specific issues of functional or non-functional requirements testing, like performance, reliability or security testing of mobile applications.

As an example, She at al. [21] have proposed a tool for testing J2ME mobile device applications that comprises a framework for writing tests using XML and a distributed run-time for executing tests automatically on the actual device, rather than on device emulators. Satoh [19, 20] presented a framework providing an application-level emulator for mobile computing devices that enables application-level software to be executed and tested with the services and resources provided through its current network.

As to the performance testing, Kim et al. [9] describe a method and a tool based on JUnit for performance testing at the unit level of mobile applications implemented in the J2ME environment.

As to the techniques for testing the correctness of a mobile application, Delamaro et al. [5] proposed a white-box testing technique that derives test cases using structural testing criteria based on the program Control-Flow-Graph. This technique is supported by a test environment that provides facilities for generating, running the tests and collecting the trace data of a test case execution from the mobile device.

More recently, a black-box testing technique for GUI Adaptive Random Testing has been presented in [11]. This technique considers two types of input events to a mobile application, namely user events fired on the application GUI, and environmental events produced by the mobile device equipments like GPS, bluetooth chips, network, etc. or by the other applications. Test cases are defined as event sequences composed by pools of randomly selected events. The technique has been experimented with six real-life applications running on Android 1.5 Mobile OS.

In the Android development platform, several tools, APIs and frameworks have been recently proposed for supporting application testing.

The Android Testing framework, besides native JUnit classes and API, includes an API that extends the JUnit API with an instrumentation framework and Android-specific testing classes. As an example, the extensions to the JUnit classes include Assertion classes (that contain specific assertions about Views and Regular Expressions), MockObject classes (that can be used to isolate tests from the rest of the system and to facilitate dependency injection for testing), and specific TestCase classes that allow peculiar components of the Android application (such as Activity, Content Provider, and Intent) to be tested in an effective manner.

Among the available tools, Monkey [16] is a built-in application that can send random event sequences targeted at a specific application and can be used for stress testing. However, pure random testing, although simple and fully automatic, may not be effective for detecting a fault. The Monkey Runner tool [17] vice-versa provides an API for writing programs (written in Python) that control an Android device or emulator from outside of Android code. Monkey Runner can be used both for functional testing, where the tester provides input values with keystrokes or touch events, and view the results as screenshots, and for regression testing (Monkey Runner can test application stability by running an application and comparing its output screenshots to a set of screenshots that are known to be correct).

The Google Code site presents the Robotium framework [18] based on JUnit that can be used to write automatic black-box test cases for testing Android applications at function, system and acceptance level.

Using Robotium, test case results can be checked by means of GUI assertions like in Web application testing using the Selenium framework.

7. Conclusions

In this paper a technique for automatic testing of Android mobile applications has been proposed. The technique is inspired to other EDS testing techniques proposed in the literature and relies on a GUI crawler that is used to obtain test cases that reveal application faults like run-time crashes, or that can be used in regression testing. Test cases consist of event sequences that can be fired on the application user interface.

At the moment, we have not considered other types of events that may solicit a mobile application (such as external events produced by hardware sensors, chips, network, or other applications running on the same mobile device) and just focused on user events produced through the GUI. In future work, we intend to propose a strategy for considering other types of events, too, in the test case definition process.

The proposed testing technique aims at finding runtime crashes or user-visible faults on modified versions of the application. In order to detect runtime crashes, at the moment, we instrument the source code of the application under test. However, in the future we plan to overcome this limitation by defining a technique that allows the crawler and the test cases to be run on the build of the Android application directly.

In the paper we just discussed an example of using the technique for testing a small size Android application, and showed the usability and effectiveness of the technique and supporting tool.

In future work, we plan to carry out an empirical validation of the technique by experiments involving several real world applications with larger size and complexity, with the aim of assessing its cost-effectiveness and scalability in a real testing context.

Moreover, in order to increase the effectiveness of the obtained test suites we intend to investigate further and more accurate techniques for the crawler to generate several kinds of input values, including both random and specific input values depending on the considered type of widget. In addition, solutions for managing test case preconditions and postconditions related to persistent data sources (such as files, databases, Shared Preferences objects, remote data sources) will be looked for.

References

[1] D. Amalfitano, A. R. Fasolino, P. Tramontana, Rich Internet Application Testing Using Execution Trace Data, Proc. of Second International Workshop on TESTing Techniques & Experimentation Benchmarks for Event-Driven Software (TESTBEDS 2010), IEEE CS Press, pp. 274- 283

[2] Android Developers. The Developer's Guide. Available at: http://developer.android.com/. Last accessed Jan. 08, 2011

[3] Android Emulator, available at: http://developer.android.com/guide/developing/tools/emulator.html Last accessed January 8, 2011

[4] F. Belli, C.J. Budnik1 and L. White. Event-based modelling, analysis and testing of user interactions: approach and case study. Softw. Test. Verif. Reliab. 2006; Wiley Eds.; 16:3–32

[5] M. E. Delamaro, A. M. R. Vincenzi, and J. C. Maldonado. A strategy to perform coverage testing of mobile applications. In Proceedings of the 2006 international workshop on Automation of software test (AST '06). ACM, New York, NY, USA, 118-124.

[6] Gartner Newsroom. Gartner Says Android to Become No. 2 Worldwide Mobile Operating System in 2010 and Challenge Symbian for No. 1 Position by 2014. Available at: http://www.gartner.com/it/page.jsp?id=1434613 Last accessed Jan. 08, 2011

[7] D. Gavalas and D. Economou. Development Platforms for Mobile Applications: Status and Trends. IEEE Software, Volume: 28, Issue: 1 , 2011, pag. 77- 86.

[8] Junit. Resources for Test Driven Development. Available at: http://www.junit.org, accessed Jan. 08, 2011

[9] 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

[10] Z. Liu, X. Gao, X.Long. Adaptive Random Testing of Mobile Application. Proc. of 2010 2nd International Conference on Computer Engineering and Technology (ICCET), IEEE Comp. Soc. Press, pp. 297-301

[11] A. Marchetto, P. Tonella and F. Ricca. State-Based Testing of Ajax Web Applications. Proc. of 2008 Int. Conf. on Software Testing, Verification and Validation, IEEE CS Press, pp. 121-130, 2008

[12] A. Memon, L. Banerjee, A. Nagarajan. GUI ripping: reverse engineering of graphical user interfaces for testing. Proceedings of the 10th Working Conference on Reverse Engineering (WCRE 2003), 2003, IEEE CS Press, pp.260 – 269

[13] A. M. Memon, Q. Xie. Studying the Fault-Detection Effectiveness of GUI Test Cases for Rapidly Evolving Software. IEEE Transaction on Software Engineering, 2005, Vol. 31, No. 10, pp. 884-896

[14] A. M. Memon and Qing Xie. Designing and comparing automated test oracles for GUI-based software applications. ACM Transactions on Software Engineering and Methodology, ACM Press, vol. 16, no. 1, 2007,.

[15] A. Mesbah, A. van Deursen. Invariant-based automatic testing of AJAX user interfaces. Proc. of International Conference on Software Engineering (ICSE 2009), IEEE CS Press, pp. 210-220, 2009

[16] Android Developers. UI Application Exerciser Monkey Available at: http://developer.android.com/guide/developing/tools/monkey.html. Last accessed Jan. 08, 2011.

[17] Android Developers. Monkeyrunner. Available at: http://developer.android.com/guide/developing/tools/monkeyrunner_concepts.html. Last accessed Jan. 08, 2011
[18] Google Code. Robotium. Available at: http://code.google.com/p/robotium/ Last accessed on Jan. 08, 2011
[19] I. Satoh. A Testing Framework for Mobile Computing Software. IEEE Trans. Softw. Eng. 29, 12 (December 2003), pp. 1112-1121.
[20] I. Satoh. Software testing for wireless mobile application. IEEE Wireless Communications, pp. 58-64, Oct. 2004
[21] S. She, S. Sivapalan, I. Warren. Hermes: A Tool for Testing Mobile Device Applications. Proc. of 2009 Australian Software Engineering Conference, IEEE Comp. Soc. Press., pp. 123-130
[22] A.Wasserman. Software Engineering Issues for Mobile Application Development. Proc. of the FSE/SDP workshop on Future of software engineering research, FOSER 2010, IEEE Comp. Soc. Press, pp. 397- 400