Using GUI Ripping for Automated Testing of Android Applications

Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, and Salvatore De Carmine
Dipartimento di Informatica e Sistemistica,
Università Federico II Napoli, Napoli, Italia
{domenico.amalfitano,anna.fasolino,
porfirio.tramontana} @unina.it,
salvatore.de.carmine@alice.it

Atif M. Memon
Department of Computer Science
University of Maryland
College Park, Maryland, USA
atif@cs.umd.edu

ABSTRACT
We present *AndroidRipper*, an automated technique that tests Android apps via their Graphical User Interface (GUI). *AndroidRipper* is based on a user-interface driven *ripper* that automatically explores the app's GUI with the aim of exercising the application in a structured manner. We evaluate *AndroidRipper* on an open-source Android app. Our results show that our GUI-based test cases are able to detect severe, previously unknown, faults in the underlying code, and the structured exploration outperforms a random approach.

Categories and Subject Descriptors
D.2.5 [**Testing and Debugging**]: Testing tools

General Terms
Reliability, Verification.

Keywords
Testing Tools, Android, Testing Automation.

1. INTRODUCTION
According to Gartner, Inc., the mobile phone/tablet Android operating system accounted for 52.5% of smartphone sales in the third quarter of 2011, more than doubling its market share from the third quarter of 2010 [8]. Moreover, in December 2011, Android Market exceeded 10 billion app downloads with a growth rate of one billion app downloads per month [4]. These numbers show the great success of this platform and indicate the necessity for cost-effective approaches for Android app development. According to Wasserman, an important software engineering challenge with mobile application development is that of finding effective solutions for achieving non-functional qualities in mobile applications and defining suitable techniques and tools to support their testing [17].

Android application testing represents a challenging activity, with several open issues, specific problems, and questions. For example, most developers remain largely unfamiliar with the Android development platform, leaving their applications prone to new kinds of bugs. Although Android apps are developed using Java technologies, they differ from standard Java client-server applications and traditional event-based desktop applications. The structure of Android apps centers instead around particular software components offered by the Android Development Framework, such as Activity, Service, Content Provider, etc., which require specific management rules and a particular lifecycle [3]. A description of typical bugs encountered in real Android applications [7] shows that frequent bugs are due to incorrect management of the 'Activity' component lifecycle. This component provides crucial functions for the application's user interface [3] and reacts to events generated by users and other system components. Incorrect management of these events often results in wrong or unsatisfactory application behavior.

We present *AndroidRipper*, an automated technique implemented in a tool that tests Android apps via their Graphical User Interface (GUI). We leverage results of recent work on model-based GUI testing. Some of these models include Event-Sequence Graphs [5], Event-Interaction-Graphs [12], Event-Flow Graphs [13], and Finite State Machines [1, 10]. Testing techniques based on these models perform test generation as a post-model creation step. The biggest obstacle to adopting these techniques for the Android platform is model development [10]. While researchers have developed techniques to reverse engineer (or *rip* [11]) some models from the subject system by fully or partially automated analysis techniques [1, 11], fully automatic analysis remains challenging for Android GUIs.

AndroidRipper extends previous work on ripping. Its goal is not to develop a model of the app. Instead, it uses ripping to automatically and systematically traverse the app's user interface, generating and executing test cases as new events are encountered. Test cases are composed of sequences of events "fireable" through the widgets of the app's GUI. Test case generation is based on the automatic dynamic analysis of the GUI that is executed in order to find and fire events in the GUI. Crucial aspects of any GUI dynamic analysis technique include: the way and order GUI events are found and fired, pre-conditions of the application and of its running environment at the time events are fired, the criterion used to stop the exploration of a given GUI, the app's initial state, and so on. Depending on the specific GUI analysis options, test cases with different fault-detection capabilities are obtained. Consequently, *AndroidRipper* is based on a configurable GUI analysis technique, performed by a GUI ripper whose behavior can be tuned, via some parameters, according to the specific application under test and specific

ASE'12, September 3–7, 2012, Essen, Germany

testing aims. *AndroidRipper*, while exploring the GUI, detects run-time crashes of the application.

We evaluate the effectiveness of different test suites, output as a result of various parameter settings, generated by *AndroidRipper*. We compare the test suites with respect to their capability to detect faults and cover code for an open-source Android app called WordPress. Our results show the feasibility and cost-effectiveness of the overall approach. Moreover, we compared our technique against the random testing approach implemented by Monkey, from the Android Development Tools. Our experimental data showed that *AndroidRipper* is more effective than Monkey in detecting failures and covering code.

2. ASSESSMENT OF CURRENT ANDROID TESTING TECHNIQUES

Android testing techniques should be able to reveal observable failures in software applications. Besides the traditional failures due to application logic bugs, Android applications often show failures that are specific of their development platform. Some specific Android bugs are reported in the classification proposed by Hu et al. [7]. The classification includes Activity, Event, Dynamic type, API, I/O, and Concurrency errors, as well as unhandled exceptions. An Android-specific testing technique is the one proposed by the same authors of the bug classification [7]. The technique is event-based and focuses on Activity, Event and dynamic type errors. The test case generation centers on the Activity components, as they provide the main entry points and control-flow drivers in Android applications. Test case generation exploits the built-in Monkey application within the Android mobile OS. Monkey [14] generates random or deterministic sequences of events automatically and can support the interaction with the mobile device. Tracing log files produced by test case executions are automatically analyzed to detect potential bugs by looking for known error patterns, including activity, event or dynamic type bugs.

Android testing has also been approached by model-based testing techniques that first obtain a formal model describing the application at a level of detail necessary for automatic test case generation. Test case generation algorithms process the model in systematic ways to produce test cases. To obtain the application's model, these techniques usually require detailed static or dynamic analysis of the application. A model-based approach for Android GUI testing has been proposed by Takala et al. [16]. The technique describes the GUI of an Android application by state machines, a very common model for representing GUIs. In order to cope with the complexity of state machines representing real-size applications, each individual view of the GUI is split into two levels as specified by two separate state machines: an action machine (describing high-level functionalities using action words and state verifications) and a refinement machine (describing action words and state verifications using keywords). These models, which must be manually generated, are used to define test cases that can be executed by a test automation tool.

An alternative approach for automatically testing an Android application by its GUI has been proposed by Amalfitano et al. [2]. The approach is based on a tool that explores the application GUI by simulating real user events on the user interface and reconstructs a GUI tree model. The nodes of the tree represent individual user interfaces in the Android application, while edges describe event-based transitions between interfaces. The GUI exploration technique supports the automatic derivation of test cases that can be executed both in crash testing and regression testing processes. In contrast to the testing technique presented by Takala et al. [16], the one proposed by [2] reconstructs the GUI model automatically and thus provides a suitable solution for GUI testing automation. However, the exploration technique used for deriving test cases in [2] is pre-defined and it is not possible to vary it in order to satisfy specific exploration needs.

Liu et al. propose a black-box approach for testing mobile applications that mixes elements of event-based testing and random testing [9]. The technique extends the Adaptive Random Testing [6] to the automatic test case generation for mobile applications. Test cases are composed by sequences of both user events and context events that come from the physical context of the device (such as GPS, compass, or other device sensors) or from other ones, like chat friends, the current activity of a user, etc.. Test cases are generated randomly by a monkey robot. This technique exploits a new definition of test case distance for mobile applications in order to spread the randomly generated test cases as evenly as possible. The experimental results show that this technique is superior to pure random test case generation in terms of earlier detection of failures.

3. DESIGN OF *AndroidRipper*

The GUIs of Android applications provide a hierarchical, graphical front-end to the application that accept as input user-generated and system-generated events from a fixed set of events and produce graphical output. Each GUI contains graphical objects; each object has a fixed set of properties. At any time during the execution of the GUI, these properties have discrete values, the set of which constitutes the state of the GUI [13]. In Android, GUIs are implemented by two main components from the development framework namely Activities and Views.

AndroidRipper dynamically analyses the application's GUI with the aim of obtaining sequences of events fireable through the GUI widgets. Each sequence provides an executable test case. During its operation, *AndroidRipper* maintains a state machine model of the GUI, which we call a GUI Tree. The GUI Tree model contains the set of GUI states and state transitions encountered during the ripping process. The ripping technique is iterative and relies on the following concepts:

- An *event* is a user action performed on a GUI widget. Events can be distinguished between data input events (such as filling in an editable text) and command input events (such as clicking on a button);
- An *action* consists of a sequence of zero or more data input events followed by a single command input event;
- A *task* is a couple (action, GUI state) representing an action performed in a GUI state; a task is executed by preliminarily reaching the GUI state and then performing the action; a task list is a set of tasks;
- The *GUI exploration criterion* is a logical predicate (composition of conditions) that establishes if the exploration of a given GUI must be continued (true value) or stopped (false value). As an example, given a GUI, a condition may evaluate the equivalence of its state with the one of already visited GUIs, or that the Depth of the resulting GUI Tree is less than a given maximum value (Maximum Depth of GUI Tree).

AndroidRipper design is based on executing tasks in a task list, initialized with tasks that are fireable in an initial GUI of the application, while the GUI Tree just contains a single state (representing the initial GUI shown by the application when the ripper starts exercising it). The task list is iteratively updated with new tasks defined from the current GUIs, and new states and state transitions are added to the GUI Tree. The GUI exploration strategy of *AndroidRipper* can be tuned varying some parameters such as the time delay between consecutive fired events, input values, GUI traversal strategy, GUI exploration criterion, etc…

4. DEMONSTRATION OF *AndroidRipper*

We implemented the *AndroidRipper* using the Robotium Framework [15] and by the Android Instrumentation class [3]. Further details about this tool and some examples of using it for crash testing real Android applications are available at a Wiki Web Page [18]. We used our implementation of the *AndroidRipper* to test an open-source Android app called "*Wordpress for Android*" (available at http://android.wordpress.org/). It provides an interactive client for creating, updating and managing blogs saved on a server. Its rich user interface allows users to write new posts, edit post content, and manage comments of blogs with built-in notifications. The app is under active development and has a broad user community, as evident by its publicly available web site. Its developers employ an issue tracking system for software development projects available at https://android.trac.wordpress.org/, containing bug tracking and linking to application version history (available at http://android.svn.wordpress.org/). We chose to analyze release *r394*, which was the newest release available at the time this work was performed. This release's source code consists of 6 Java packages, a total of 71 files containing 334 classes and 1,489 methods; in all, 10,017 executable lines of code.

Using *AndroidRipper's* settings discussed earlier, we tested the app and measured a number of metrics in order to assess both effectiveness and costs of the technique [12]. We counted the number of bugs detected (Metric M0) and the number of crashes occurred at run-time (metric M1) for measuring the *Defect Detection Effectiveness* of the technique. Moreover, we measured *Code coverage* that provides an evidence of the technique's potential ability in fault detection: the better the code coverage, the better the potential goodness of a testing technique. We used the Number of LOCs covered / Total number of executable LOCs) % (metric M2) as coverage metric. Lastly, we assessed *process cost* by resources spent by testing. We used the time (in hours) spent for GUI ripping (M3). The testing sessions were all executed using a workstation equipped with a Windows XP Professional O.S., with 2 GB RAM and a Dual Intel Pentium E2200 at 2.2GHZ.

Because WordPress is a client-server application, we had to control its state too. In a first round, called R1, the app was tested in the client side precondition called *No Login* (NL), where the user installed the application for the first time and accepted the disclaimer. No specific preconditions were set for the server side, because they were irrelevant. This session lasted about 12 minutes, due to the very restrictive precondition of the application that did not allow the ripper to explore the app GUI further. The ripper did not record any crash of the app and code coverage was very low (just 2,65 LOC coverage %).

In the second round, called R2, the client-side precondition was set to *First Login* (FL) where the user had previously installed the application on the mobile device for the first time, accepted the disclaimer and correctly logged in. The server side precondition was *More than one Blog* (MB), where the WordPress database was initialized to two blogs, both having two pages, six comments, two posts with multimedia elements (one of which is 'Hello World'), one tag, and one comment. With these new preconditions, the ripper was able to cover more than 39% of LOC of the app in less than 5 hours and recorded a considerable number of crashes (6 crashes) that were not documented by the app Web page. Using the WordPress bug track system, we reported (with the 'AndroidCrawler' user name) these crashes to the app developers by opening a ticket. The developers fixed the bugs. By analysing the change-set made for correcting bugs, we recognized that crashes were due to 3 distinct bugs of the applications (namely, B1, B2, and B3). We classified the bugs as per a standard classification scheme [7]: (1) *Concurrency* (C), errors due to the interaction of multiple processes or threads, and (2) *Others* (O), due to incorrect application logic implementation. The first three rows of Table 1 report for each bug a short description of the crash, the classification of the bug, the Java exception, and the corresponding http addresses of ticket and change-set from the bug tracking system of Wordpress.

In the third round, R3, the client side precondition was FL and the server side was *Single Blog* (SB), the state in which the WordPress database contains a single new blog with two pages, six comments, two posts (one of which is the auto-generated 'Hello World' post), no multimedia element, one tag, and one comment. With these new preconditions, the ripper was able to cover about 38% of LOC of the app in less than 5 hours and found 8 crashes. Six crashes were due to the three bugs detected earlier. Two crashes were completely new to the app developer and were attributed to a new bug (B4). The changeset analysis revealed that bug B4 may be classified as an *Activity* error (A), due to incorrect management of the Activity lifecycle, specific to Android apps. Further details about this bug are reported in row 4 of Table 1 and the results are summarized in Table 2.

In order to compare the results achieved by our GUI ripping technique against the ones reachable by other test automation solutions currently available, we performed another testing session using Monkey tool. Monkey is a tool for random stress and crash testing of Android GUIs [14]. To the best of our knowledge, this is the only non-commercial tool for Android automated testing available. Monkey is able to fire random user events on the GUI of an app and stops the exploration when a given input number of events were fired. We tried several configurations of Monkey, with various GUI exploration options. Among them, we report the results of a single execution of Monkey (RM) whose time duration was 4.46 hours, hence, comparable with the duration of R3 and R2 sessions. In the considered execution, Monkey had to fire 45,000 events with the default value of event type statistical distribution. In this experiment Monkey found 3 crashes corresponding to bug B2 and reached 25.27 % LOC coverage, being less effective than the ripping based testing sessions R3 and R2. These results are reported in the last column of Table 2.

5. CONCLUSIONS

We presented *AndroidRipper*, a technique based on GUI ripping for automatic testing of Android applications. Our evaluation using the "WordPress for Android" application revealed four undocumented bugs, automatically detected in less than five hours. This datum shows the effectiveness of the technique in finding real bugs and its suitability for testing processes that need to be carried out in a short amount of time. Moreover, the experimental data showed that the proposed technique is more effective in bug detection than the random testing technique implemented by Monkey.

Table 1: Crash Descriptions of bugs detected

Id	Crash Description	Bug Class.	Java Exception	Ticket and Changeset
B1	The app crashes trying to opening the default post "Hello World"	O	StringIndexOutOfBoundException	https://android.trac.wordpress.org/ticket/206 https://android.trac.wordpress.org/changeset/398
B2	The app crashes when the Stats activity is rapidly opened and closed (via the Back key).	C	BadTokenException	https://android.trac.wordpress.org/ticket/208 https://android.trac.wordpress.org/changeset/420
B3	The app crashes when the Stats activity is open and the Refresh button is clicked while the progress bar widget is still loading.	C	NullPointerException	https://android.trac.wordpress.org/ticket/212 https://android.trac.wordpress.org/changeset/423
B4	The app crashes when the user opens a post and tries to share it within his blog. The crash occurs when there is a single blog in the app.	A	NullPointerException	https://android.trac.wordpress.org/ticket/218 https://android.trac.wordpress.org/changeset/446

Table 2: Bugs, Coverage and Cost Data

	R1	R2	R3	RM
# Crashes of Bug B1		4	4	
# Crashes of Bug B2		1	1	1
# Crashes of Bug B3		1	1	
# Crashes of Bug B4			2	
Total Bugs	0	3	4	1
Total Crashes	0	6	8	3
% Covered LOCs	2,65	39,32	37,83	25.27
Time (hours)	0.2	4.88	4.58	4.46

6. REFERENCES

[1] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. 2008. Reverse Engineering Finite State Machines from Rich Internet Applications. In Proceedings of the 2008 15th Working Conference on Reverse Engineering (WCRE '08). IEEE Computer Society, USA, 69-73.

[2] D. Amalfitano, A. R. Fasolino and P. Tramontana, A GUI Crawling-Based Technique for Android Mobile Application Testing, Third International Workshop on TESTing Techniques & Experimentation Benchmarks for Event-Driven Software, IEEE CS Press, pp. 252- 261.

[3] Android Developers. The Developer's Guide. http://developer.android.com/guide/, last accessed on February 29th, 2012

[4] Eric Chu. 2011. 10 Billion Android Market Downloads and Counting, http://android-developers.blogspot.com/2011/12/10-billion-android-market-downloads-and.html last acc. on February 29th, 2012

[5] Fevzi Belli, Christof J. Budnik, and Lee White. 2006. Event-based modelling, analysis and testing of user interactions: approach and case study: Research Articles. Softw. Test. Verif. Reliab. 16, 1 (March 2006), 3-32.

[6] Tsong Yueh Chen, Fei-Ching Kuo, Robert G. Merkel, and T. H. Tse. 2010. Adaptive Random Testing: The ART of test case diversity. J. Syst. Softw. 83, 1 (January 2010), 60-66.

[7] Cuixiong Hu and Iulian Neamtiu. 2011. Automating GUI testing for Android applications. In Proceedings of the 6th International Workshop on Automation of Software Test (AST '11). ACM, New York, NY, USA, 77-83.

[8] Gartner. 2011 Gartner Says Sales of Mobile Devices Grew 5.6 Percent in Third Quarter of 2011;http://www.gartner.com/it/page.jsp?id=1848514 last acc. on February 29th, 2012

[9] Zhifang Liu, Xiaopeng Gao and Xiang Long. 2010. Adaptive Random Testing of Mobile Application. In Proceedings of the 2nd International Conference on Computer Engineering and Technology (ICCET '10), IEEE Computer Society, Washington, DC, USA, 2, 297-301.

[10] Alessandro Marchetto, Paolo Tonella, and Filippo Ricca. 2008. State-Based Testing of Ajax Web Applications. In Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation (ICST '08). IEEE Computer Society, Washington, DC, USA, 121-130.

[11] Atif Memon, Ishan Banerjee, and Adithya Nagarajan. 2003. GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. In Proceedings of the 10th Working Conference on Reverse Engineering (WCRE '03). IEEE Computer Society, Washington, DC, USA, 260-269.

[12] Atif Memon and, Qing Xie. 2005. Studying the Fault-Detection Effectiveness of GUI Test Cases for Rapidly Evolving Software. IEEE Trans. Softw. Eng. 31, 10, 884-896.

[13] Atif M. Memon. 2007. An event-flow model of GUI-based applications for testing: Research Articles. Softw. Test. Verif. Reliab. 17, 3 (September 2007), 137-157.

[14] Android Developers, The Developer's Guide. UI/Application Exerciser Monkey, http://developer.android.com/guide/developing/tools/monkey.html last accessed on February 29th, 2012

[15] Robotium. http://code.google.com/p/robotium/, last accessed on February 29th, 2012

[16] Tommi Takala, Mika Katara, and Julian Harty. 2011. Experiences of System-Level Model-Based GUI Testing of an Android Application. In Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation (ICST '11). IEEE Computer Society, Washington, DC, USA, 377-386.

[17] 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

[18] Android GUI Ripper Wiki, available at: http://wpage.unina.it/ptramont/GUIRipperWiki.htm, last accessed on July 8th, 2012.