2012 28th IEEE International Conference on Software Maintenance (ICSM)

A Toolset for GUI Testing of Android Applications

Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, Gennaro Imparato
Dipartimento di Informatica e Sistemistica, Università di Napoli Federico II,
Via Claudio 21, 80125 Napoli, Italy
{domenico.amalfitano anna.fasolino, ptramont}@unina.it, salvatore.de.carmine@alice.it, genn.imparato@studenti.unina.it

***Abstract*— This paper presents a toolset for GUI testing of Android applications. The toolset is centered on a GUI ripper that systematically explores the GUI structure of an application under test with the aim of firing sequences of user events and exposing failures of the application. The toolset supports the execution of a testing procedure that automatically performs crash testing of subject applications and provides test results made of several artifacts. The paper illustrates some examples of using the toolset for testing real Android applications.**

***Keywords- Android application testing, GUI ripping, testing automation*[1]**

I. INTRODUCTION

GUI ripping is a reverse engineering technique that aims at reconstructing a GUI model of an existing software application by dynamically interacting with its user interface [6]. A GUI provides a hierarchical, graphical front-end to the application. It is usually implemented as an event-based system that accepts as input user-generated events and produces 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 [7]. A GUI ripping technique can be implemented by a tool, called GUI ripper, that iteratively fires events on the GUI and observes the resulting GUI state changes. The observed behavior can be represented by suitable models, such as finite state machines, event-flow graphs, or GUI trees.

A GUI ripper can be used to execute automated software testing processes too: in this case the ripper is used as an engine that fires sequences of events on the GUI of the application under test (AUT) with the aim of searching for application failures. Depending on its 'business intelligence', the ripper either can behave like a 'stupid' robot that randomly fires events on the GUI (even firing the same event more times), or it may be able to explore the GUI according to a systematic, well-defined GUI traversal strategy (such as breadth-first, depth-first, or others). For the aims of GUI testing [7], the ripper should be able to exercise the application when it is in several pre-conditions and using sequences of events that are able to emulate the wider set of possible human behaviors. GUI ripping has been used with success for GUI testing of both traditional desktop applications and Web applications [8]. One contribution of our work consists of porting GUI ripping to the context of mobile application testing. In particular, we explored the feasibility of using GUI Ripping for testing automation of mobile Android apps, that is now a relevant research challenge. Android operating system is becoming indeed the most diffused OS for smart-phones and tablets [4, 10], with over 450,000 apps on the Android market on June 2012. Defining effective and possibly automated approaches for Android testing is now a key challenge for the software engineering community [12].

In [1] we presented a technique based on GUI ripping for testing Android applications. According to the experimental results we obtained, that technique provides a viable and effective solution for improving the reliability of applications that often present quality issues due to very rapid and ad hoc development approaches, with very frequent releases of new application updates. The testing technique is executable with a reduced manual intervention of a tester, thanks to a GUI ripper that automatically explores the GUI and detects run-time crashes of the application. This technique overcomes some limitations of currently available Android testing tools and technologies coming either from the Android SDK [2] (such as Monkey and MonkeyRunner), either from Google code projects like Robotium [9], or from other research projects, such as [11]. Most of these solutions indeed do not provide completely automated testing approaches, but rather they require a considerable manual intervention or testing programming.

In order to further extend the automation level of this testing technique, we have developed a set of tools that allow the tester manual intervention in the testing process to be considerably reduced. In this paper we present this toolset showing some implementation details about the tools and how they can be used. The paper is organized as follows: Section II presents the GUI Ripper tool, while Section III describes the testing procedure based on the Ripper and the supporting testing tools. In Section IV some experimental results of using the toolset for crash testing of real Android applications are provided. The paper finally presents some conclusions and future work.

II. THE ANDROID GUI RIPPER TOOL

A GUI ripper must be able to interact autonomously with a running application through its GUI and to observe the

1 This work has been co-funded by the Italian Ministry of Education, University and Research, within the initiative of Operational Research and National Competitiveness Programme, PON REC 2007-2013, in the context of the research project IESWECAN

978-1-4673-2312-3/12/$31.00 ©2012 IEEE

resulting application behavior. In our research project on Android testing automation, we decided to develop a GUI ripper satisfying the following requirements too:

- The ripper's GUI exploration strategy has to be highly customizable: the tester can decide which type of events to fire on the GUI, the subset of GUI graphical objects on which to fire events, the way events will be fired, the GUI traversing strategy (e.g. depth first, breadth first, etc.), the termination criterion used to stop the GUI traversal, etc.
- The ripper has to be highly *evolvable*, in order to allow new exploration strategies to be implemented with a reduced impact on the overall ripper architecture.
- The ripper must be able to deal with *efficiency* issues related to the well known problem of GUI interface states and events explosion. To this aim, the ripper will have to adopt suitable solutions to stop the exploration of a given interface, such as the ones based on state equivalence criteria that allow the infinitely many states of a GUI to be partitioned into a finite set of equivalent 'abstract' states. Two states will be considered equivalent if the same events can be fired on the same set of widgets (more details about this criterion are reported in [1]).

To satisfy these requirements, we designed a modular GUI ripper whose software architecture is composed of nine main components, named Scheduler, Robot, Abstractor, Extractor, Engine, Strategy, Planner, Comparator and Persistence Manager, respectively. Figure 1 shows these components and how they are interconnected.

The Engine component implements the main business logic of the ripper. Its role is to launch an iterative ripping process that starts from an initial GUI of the AUT (hereafter, initial state). At each iteration, the Engine uses the Robot to emulate the user interaction with the GUI and the Extractor to get the current state of the GUI. The Extractor and the Abstractor components cooperate to define an 'abstract state' to be associated with the current GUI state. The current GUI state description is then sent to the Planner to select a set of events fireable in the GUI. The Planner transforms these events into executable tasks (i.e. sequences of legal events, from the initial state of the GUI) that will be stored in a task list managed by the Scheduler component. The Scheduler component is responsible for establishing the tasks' execution ordering according to the chosen GUI exploration strategy and provides the Engine with the next task to be executed by the Robot.

After the task execution, the Engine delegates the Strategy to establish whether the GUI exploration can stop, according to the ripping termination criterion, or not. The evaluation of the termination criterion may require an additional component, the Comparator, which assesses the equivalence between the current GUI state and other ones previously defined. The iterative ripping process proceeds until the task list is empty. During the process, the Persistence Manager component is invoked to store the partial results of GUI exploration to disk. At the end of the ripping process, the overall results of the ripping session are exported in an open format output file and saved to disk.

As to the ripper implementation details, we had to find a suitable solution for the ripper to interface the running Android application, that is implemented in Java. In Android, for protection aims, no application has permission to perform any operations that would adversely impact other applications. To overcome the limitation, we decided to implement the Ripper as an Android JUnit test case. This solution allows the ripping program to be run in the same process of the AUT, sharing with it the same execution environment. The ripper exploits the "hooks" in the Android system provided by the Android Instrumentation Framework [3] to interact with the GUI of the application under test, and the JUnit assertion mechanism for failure detection. Moreover, the ripper uses the Robotium framework too [9], since it provides powerful APIs for accessing the GUI structure and to interact with it at runtime. In particular, by means of reflection mechanisms it is possible to extract a list of the widgets composing a GUI and of the related fireable events. Actually the Ripper test case doesn't define any test unit to execute specific functionalities of the AUT, but rather is able to dynamically generate and execute "tentative test units" (the tasks, e.g., sequences of user events) by inspecting the properties of the GUI.

The ripper final outputs include (1) a representation of the explored GUI, called GUI Tree, that is a graph whose vertexes represent the states of the GUI and edges provide transitions between consecutive states and (2) a crash report file, providing details about each crash of the Android application observed during ripping.

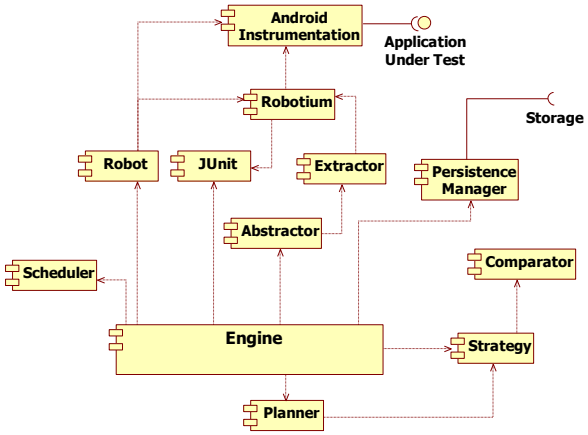

Figure 1. Software architecture of the Ripper

III. THE GUI TESTING PROCESS AND THE SUPPORTING TOOLSET

In order to use the GUI Ripper in Android testing processes, we have developed an automated test procedure that provides a solution to the following questions:

1) obtaining a test program composed of Ripper and AUT code interpretable by the Dalvik Virtual Machine (i.e.

a Java Virtual Machine specific for Android platform) and deploying them on a target Android Virtual Device (AVD), that is an Android device emulator;

2) saving a state of the AVD (i.e. the test pre-conditions of the AUT) that will represent the pre-conditions of each ripping task execution (since a task is a sequence of events from a starting GUI state);

3) restoring the AVD initial state, at the start of each ripping task execution;

4) launching single ripping tasks;

5) obtaining test results by means of suitable test reports (including coverage files, bug reports, and GUI models).

The implemented procedure comprises three sequential steps, supported by specific tools. Figure 2 illustrates the overall procedure and how its steps can be performed with the tool support. The first step (called Deploying) obtains the testing program executable on the AVD. In this step, the code of the application under test may be instrumented for the aim of obtaining code coverage measures; the Ripper options can be defined by the tester and set in the Ripper code; the testing program is built and deployed on the device emulator. Finally, the tester can interact with the AVD to take an AVD Snapshot providing an image of the initial state of the AUT that will provide the initial ripping state at each iteration. The tools employed in the first step include:

- *Source Code Instrumentation* tool, that is implemented as a batch file that automatically instruments the code of the AUT for measuring and reporting Java code coverage and builds the Instrumented Application Under Test (*IAUT.apk*). This tool exploits the EMMA libraries for Java code coverage [5].
- *Ripper Options Configuration* tool, that the tester uses for generating a XML configuration file containing the Ripper options defined by the tester (*Ripping Options.xml*).
- *Deployer* tool, that is a batch file that automatically links together the Ripper (*Ripper.apk*) and the IAUT and deploy them onto the target AVD. It deploys the XML Ripping configuration file onto the AVD, too.

The second step of the procedure is the Ripping step in which the testing program is executed on the AVD with the IAUT starting from the initial state defined by the Snapshot. Its output consists of coverage files reporting details about source code coverage of each exercised sequence of events, a crash report providing details about occurred run-time crashes due to Java unhandled exceptions, and an XML file providing an intermediate representation of the GUI Tree produced by the Ripper. As Figure 2 shows, this step is performed automatically by the *Ripper Executor* tool that executes the JUnit test case associated to the Ripper. This tool is also responsible for resetting the IAUT initial conditions before exercising each task defined by the Ripper. It has been implemented as a batch file.

The last procedure step (Post-processing step) produces a unified code coverage report, GUI models that are suitable for further model based testing techniques, and a JUnit test suite made of a set of re-executable test cases that reply the same sequences of GUI events fired by the Ripper. These test cases may be usable in regression testing activities. The tasks of the latter step are automatically performed by the following three tools illustrated in Fig. 2:

- *Code Coverage Generator*, that is a batch file that merges the partial coverage files and returns an integrated coverage report in HTML format.
- *GUI Model Translator*, that is a Java tool that produces the GUI Tree model in the file formats interpretable by graphical visualizers.
- *JUnit Test Suite Generator*, that is a Java tool responsible for translating the sequences of events fired by the Ripper into JUnit executable test cases that can be used for regression testing.

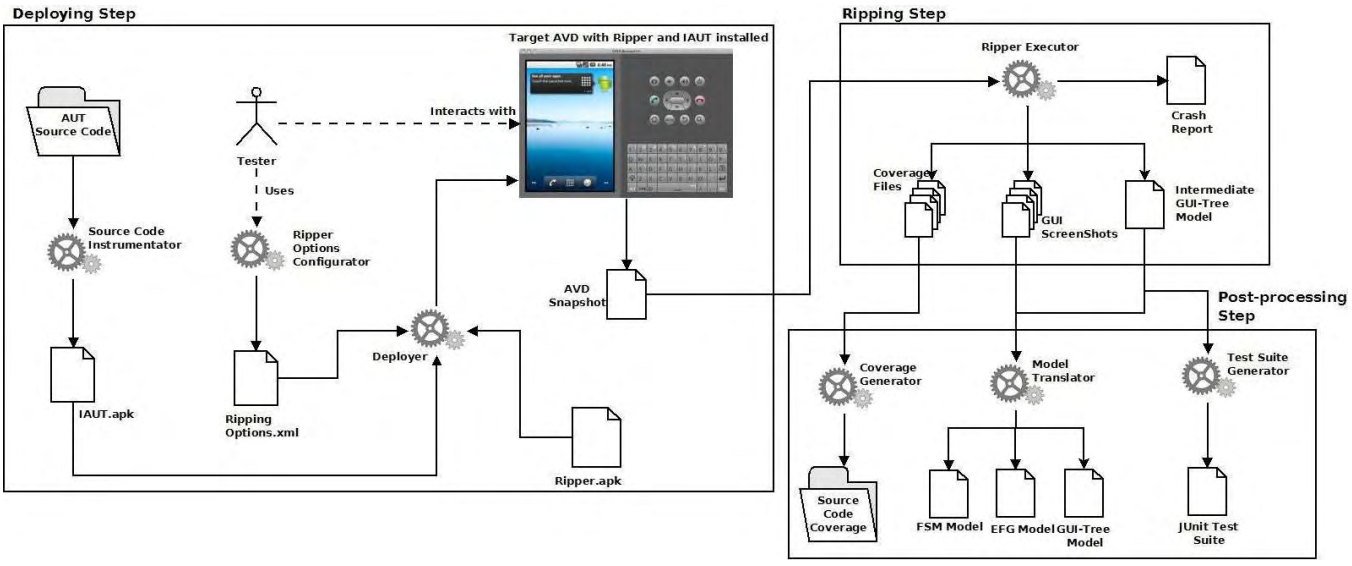

Figure 2. The GUI Testing Procedure

IV. EXPERIMENTS

This section presents the results of a study in which two real Android applications were tested using the proposed toolset. Both applications were chosen since they are downloadable from the Android market provided by Google Play and their development projects are accessible through web based hosting servers that offer a bug tracking system, too. The first app is called AardDict and is the port to Android OS of the Aard Dictionary desktop application. The second application, named TomDroid, is a note-taking application with online synchronization compatible with the Tomboy desktop application for note-taking.

Our testing approach can be performed by a tester having a basic knowledge of the application under test (in order to set its initial state) and a basic knowledge of the ripper (in order to set its parameters). The tester performed crash testing of both applications in more testing sessions where he defined different applications' pre-conditions and ripper configurations. As to the preconditions, the tester saved three AVD Snapshots of TomDroid associated to three notable states of running the app, e.g., the states of the app just installed on the device and ready to be executed for the first time, the app already configured to synchronize the notes stored on the device SD, and the app configured to synchronize the notes with the Ubuntu One Website). As to AardDict, the AVD Snapshots represented the notable states of the app with one installed dictionary and with more than one dictionary, respectively. The tester configured the ripper in order to perform GUI ripping with several time intervals between consecutive events (1 second, 2 seconds and 6 seconds, respectively), in order to emulate the behavior of both faster and slower users. Moreover, he configured the ripper either to build tasks made of sequences of at least three events, or tasks with no predefined length (but using a state equivalence criterion to stop the GUI exploration). AardDict was tested in 12 testing sessions that almost all required less than three hours to perform GUI ripping. TomDroid was tested in 18 testing sessions almost all with GUI ripping performed in less than two hours.

As to the testing results, depending on the testing session, the code coverage of AardDict varied between 57 and 69 LOC percentages, while for TomDroid it ranged between 34 and 53 LOC percentages. Moreover, a new bug (never notified to the app author before) of AardDict was found in a session and notified to the author by the bug tracking web site of the application (https://github.com/aarddict /android/issues/44). As to TomDroid, the testing found one bug that was already known to the developer (https://bugs.launchpad.net/tomdroid/+bug/902855).

These results showed the utility of the testing procedure that allowed a variety of testing sessions to be executed with a reduced tester intervention (results from other experiments are reported in [1]). Moreover, the observed variability of code coverage and fault detection measures with ripping options and app pre-conditions confirms the necessity of a flexible experimental testing environment, like the one offered by our toolset. The toolset is thus suitable for executing further experiments aiming at better assessing the relationship between ripper options and testing effectiveness.

V. CONCLUSIONS AND FUTURE WORKS

The toolset proposed in this paper is able to automate almost completely Android application crash testing. With respect to the proposed testing procedure, manual interventions of a tester are required just for configuring the GUI Ripper options and saving an AVD Snapshot. The most expensive step of the process, i.e. GUI ripping, is performed automatically, instead.

In future works we plan to implement a solution based on services (e.g. Web Services) in order to allow the remote execution of the testing process. In order to try the testing process execution on Android applications, the Wiki Web page (at http://wpage.unina.it/ptramont/GUIRipperWiki.htm) at the moment provides a set of preconfigured packages allowing the local execution of the toolset.

REFERENCES

[1] D. Amalfitano, A.R. Fasolino, P. Tramontana, S. De Carmine, and A. Memon, "Using GUI Ripping for Automated Testing of Android Applications", Proceedings of IEEE/ACM Conference on Automated Software Engineering, ASE 2012.

[2] Android developer Standard Development Kit http://developer.android.com/sdk/index.html, last acc. June 27, 2012

[3] Android Developer, Testing Fundamentals, available at: http://developer.android.com/tools/testing/testing_android.html#Instrumentation, last accessed June 27, 2012

[4] App Brain market statistics, available at http://www.appbrain.com/stats/number-of-android-apps, last accessed June 27, 2012

[5] Emma, a free Java coverage tool, available at: http://emma.sourceforge.net/, last accessed June 27, 2012

[6] 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

[7] A. Memon, Q. Xie, "Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software", IEEE Transactions on Software Engineering, , vol.31, no.10, pp. 884- 896, Oct. 2005

[8] R. Brice, S. Sampath, A. Memon, "Developing a Single Model and Test Prioritization Strategies for Event-Driven Software",IEEE Trans. On Software Engineering, Vol. 37, Issue 1, Jan 2011, pp. 48- 64.

[9] Robotium project, available at: http://code.google.com/p/robotium/, last accessed June 27, 2012

[10] StatCounter Global Stats, available at http://gs.statcounter.com/#mobile_os-ww-monthly-201105-201205, last accessed June 27, 2012

[11] T. Takala, M. Katara, J. Harty- Experiences of System-Level Model-Based GUI Testing of an Android Application. 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation, IEEE Comp. Soc. Press, pp. 377- 386

[12] A.Wasserman, Software Engineering Issues for Mobile Application Development, Proc. of the FSE/SDP workshop on Future of software engineering research, IEEE Comp. Soc. Press, pp. 397- 400