Automated GUI Testing for Android Application
Akanksha Ashok Magare¹ & Prof. Manjushree D. Laddha²

¹Computer Engineering, Dr. Babasaheb Ambedkar Technological University- Lonere,  
²Computer Engineering, Dr. Babasaheb Ambedkar Technological University- Lonere, (Indiam)

Abstract: Most of the people spend significant time on mobile phone. Due to outbreak in technology and software usage, testing of software is an important task. Graphical user interfaces (GUIs) are by far the most popular means used to interact with today’s software. As GUIs are critical components of today’s software. Moreover the increasing number of mobile applications with rich Graphical User Interfaces (GUI) creates a growing need for automated techniques of GUI Testing for mobile applications. The functional correctness of a GUI is required to ensure the safety, robustness and usability of an entire software system. The goal of GUI testing is to enhance fault detection rate, good coverage and increased efficiency. Testing is performed to improve the overall performance of GUI. The technique is based automatically builds a GUI tree of the application and obtains test cases that can be automatically executed. Our results show that GUI-based test cases are able to detect several previously unknown faults in the underlying code, sequence problem.

Keywords: Software testing, Android application Testing, Testing Automation, Test case generation, log analysis

Introduction

GUI is a program that takes advantage of graphics capabilities of computer. GUI provides use easy way to interact with software. GUI acts as interface between user and software. GUI makes the software interface easy but they also complicate the software development process. Due to sophisticated and complex GUIs there is a need of GUI testing to ensure correctness of GUI. Any complexity in software needs to be tested because untested code is potential source of bugs. GUI testing is vital to make entire system safer and more robust. GUI testing is an area of growing importance, facing a number of challenges. GUI testing is a set of activities used to test the GUI of any product to ensure that it meets its written specification. The whole system can be executed by means of GUI. GUI has many operations that need to be tested. We test GUI from different perspective which includes test coverage, test case generation, test oracle and regression testing.

Different researchers have proposed techniques keeping in view the different aspects and features of GUI. Objective of each technique is to generate test cases which are capable to detect maximum faults. GUI testing provides an assurance of proper functioning of application/software. GUI testing can be done by manual methods which are time consuming and are not so much used or by automated methods which are used now a days. Android applications can be actually considered Event Driven Software (EDS) whose behavior 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.

1. Background And Related Work

The Android Developers Web site [1] 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. 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 [1]. 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.

2.1 Open Issues with Android Application Test

Since the behavior 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, 5, 6], exploit the analysis of user session traces for deriving test cases [2], Crawlers [7] 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 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 [1] provides an integrated testing framework based on JUnit [3] 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.

2. Proposed Approach

Like the crawler-based technique presented by 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 analyzed by the crawler at run time belong to the conceptual model of an Android GUI that is represented by the class diagram.

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 of UI 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.).

Figure 1. Conceptual Model of an Android Application GUI
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.

### 3.1 Graph Converter

The Graph Converter provides a platform-independent framework to convert the GUI Tree model, output into a graph representing relationships between events in the GUI of the application. The framework provides support for processing the input GUI Tree and generating a graph which is subsequently used for test case generation.

An EFG is a directed graph representing all possible event interactions on a GUI. Each node in an EFG represents a GUI event (e.g., click-on-Create, click-on-OK). An edge from node v to node w represents follows relationship between v and w, indicating that event w can be performed immediately after event v. An EFG is analogous to a control-flow graph, in which vertices represent program statements and edges represent execution flows between the statements.

The construction of the EFG is based on the identification of modal and modeless windows. The Graph Converter leverages this information while constructing the EFG. Restricted-focus events open modal windows. Unrestricted-focus events open modeless windows.

Terminal events close modal windows. Expand events are used to reveal hidden sub-components (e.g., sub-menu items or tab elements). System-interaction events are not used to manipulate the structure of the GUI; rather, they interact with the underlying software to perform some actions.

### 3.2 Test Case Generator:

The Test Case Generator framework provides three core features that may be used by a tester to implement a specific test generator. First, depending on the model exploration strategies desired, various test case generators may be built around a single graph model. In theory, a GUI test case can be of any length, possibly infinite, as a single widget can be clicked an infinite number of times. With multiple widgets on a GUI, the possible combinations can easily grow very large. Similarly, a GUI test suite can be of any size, possibly infinite. One can construct a test suite containing test cases of all lengths.

Second, the Test Case Generator generates values for event parameters if required (such as adding values for text-input fields by reading text inputs from the configuration file).

Third, the Test Case Generator framework inserts connecting" events in the test case to make it executable on the real GUI. Connecting events may be added to the test case e1-e2, to create c1-e1-c2-e2. This insertion makes e1 accessible from the initial state of the GUI, and also makes e2 accessible from e1.

Here, the added event c1 is called a prefix" event.

### 3.3 Replayer

The Replayer provides a framework for executing a test case automatically on the AUT. The tool also provides hooks for observing and recording the AUT during test case replay. The Replayer takes as input an executable test case, the GUI Tree and the graph model, launching the application with the same initial state used during ripping. For each event in the test case, the Replayer uses the information in the GUI Tree and the graph model to identify the GUI window and widget on which the event needs to be executed. The tool invokes the event on the identified widget. When all events have executed, the Replayer closes all open windows and shuts the application down.

### 3.4 Oracle Verifier:

The Oracle Verifier provides mechanisms to determine whether a GUI executed correctly for a test case. In addition to a sequence of events to be executed, a test designer must decide both what to assert and when or how often to check an assertion, e.g., after each event in a test case or after the entire test case completes execution. Variations of these two factors significantly impact the fault-detection ability and cost of the execution and maintenance of a GUI test case. Currently, we support two Oracle Verifier implementations with the Crash Verifier for reporting crashes and the State Verifier...
for matching output GUI states across different test case executions.

Figure 2. Architecture Module

3. Technique for Testing Android Application

4.1 The GUI Crawler

The crawler builds the GUI tree implementing a depth first search using an iterative algorithm. crawler fires events on the application user interface capturing data about interfaces and events that will be used also to decide the further events to be fired. During the GUI exploration the crawler is able to perform a first crash testing. 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).

Proposed Algorithm:

1. 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;
2. Detect all the interface fire able 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 List1.

Repeat

3. Choose one fire able event E from the Event List, set the needed preconditions and fire it, according to its description.
4. 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.
5. 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 fire able events, the corresponding GUI node will be a leaf of the tree, otherwise the new interface fire able 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 fire able Event list is empty

4.2 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. [6], 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 behavior has changed and whether new faults have emerge. In the regression testing of an Android application 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 behaviors. 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 [8]. As an example, the Monkey Runner tool [9] 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 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.

5 Conclusion and Future Work

Proposed work is a test framework to make it easy to write powerful and robust white black-box test cases for Android applications. With the support of proposed work, test case developers can write function, system and acceptance test scenarios, spanning multiple Android activities. Proposed work will support Activities, Dialogs. This is new fully automatic technique to test GUI-based Android apps. The technique is based on the observation, extraction, and abstraction of the run-time state of GUI widgets. The abstraction is used to create a scalable state-machine model that, together with event-based test coverage criteria, provide a way to automatically generate test cases. The technique was demonstrated via an empirical study on 4 open-source software applications. The results showed that the test cases generated were useful at detecting serious and relevant bugs in the apps. Moreover, this study showed that the combination of model-learning with model based testing techniques is a promising approach for achieving a better fault detection capability in Android app testing. Manual interventions of a tester are required just for configuring the Input options and saving an AVD Snapshot. The most expensive step of the process, i.e. GUI crawler, is performed automatically. 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 post conditions related to persistent data sources (such as files, databases, Shared Preferences objects, remote data sources) will be looked for.

6. References


