Execution and Property Specifications for JPF-Android

Heila van der Merwe, Brink van der Merwe and Willem Visser
Dept. of Computer Science
University of Stellenbosch, South Africa
{hvdmerwe, abvdm, wvisser}@cs.sun.ac.za

ABSTRACT
JPF-Android is a model checking tool for Android applications allowing them to be verified outside of an emulator on Java PathFinder (JPF). The Android applications are executed on a model of the Android software stack and their execution driven by simulating user and system input events. This paper follows from our previous work describing the design decisions and implementation of JPF-Android. Here we discuss the syntax and implementation of the scripting environment which is used to drive the execution of the Android application under analysis. It also focuses on a further extension to the tool used to automatically monitor the runtime behavior of Android applications.

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

Keywords
Android application, Java PathFinder, Testing, Verification, Model Checking, Runtime monitoring

1. INTRODUCTION
Android applications must be robust, consistent in their behavior and always retain their application state to provide a good user experience. Currently, many verification and testing techniques are being researched to develop a framework assisting developers in creating robust and error free Android applications.

The most common way of testing Android applications is using one of the Android built-in testing tools[1]. These tools execute tests on the emulator or device running Android. Running tests on the emulator/device, however, is very slow. This is due to the fact that the test cases have to be compiled, packaged into dex byte-code, uploaded and then run on the the device. Finally, the results have to be sent back and displayed to the user. The Android software stack is also very restrictive in terms of the functionality it exposes internally to its testing frameworks as well as externally to software running outside of the emulator/device. Additionally, the Android Operating System (OS) is not easily extensible. To include testing code into the OS, a custom Android ROM has to be compiled to extend existing OS capabilities.

Verifying Android applications yields two main challenges: providing an adaptable and customizable environment that models the Android software stack on which to run Android applications, and providing an input model that can simulate input events to drive the application execution.

This paper follows from our previous paper "Verifying Android Applications using Java PathFinder" [10], which describes the design and implementation of JPF-Android. JPF-Android is a model checking tool for Android applications allowing them to be verified outside of an emulator on Java PathFinder (JPF). This paper discusses the syntax of JPF-Android's input script and how events are simulated on the application under test. The focus of this paper, however, is on an extension to the tool that can verify the execution of Android applications during runtime. Lastly, it illustrates how the tool's runtime monitoring functionality can be used by applying it to an example Android application.

2. JPF-ANDROID
Every application verified on JPF requires a *.jpf application properties file. This file defines the target of the application i.e. where JPF should start execution. For Java applications the target is the class containing the main method starting the application. Android applications do not have a main method. They are initiated by the Android system invoking callback methods, corresponding to specific lifecycle stages on the application components. Each Android application has a launcher or main Activity. This Activity serves as the main entry point for the application. JPF-Android requires the target to be set to this Activity. When this class is loaded a main method is injected into this Activity by JPF-Android. This main method starts up the entire Android software stack model which in turn starts up the application.

This properties file also contains the name of the input script file and the Checklist definition file for this project.

JPF-Android makes use of an input script containing interesting and important event sequences to drive the application under test. This script is parsed and its state managed by the scripting environment. When the Android application reaches a point in the execution where it is idle i.e. there are no more messages in the message queue, the next event is requested from the scripting environment. An example of an input script is given in Figure 3.

The Checklist definition file contains user defined Checklist definitions and method-to-Checkpoint mappings. These

[1] developer.android.com/tools/testing/testing_android.html

```
#— dependencies on other JPF projects
@using = jpf-android

#— target setup
target = za.android.vdm.rssreader.TimelineActivity
classpath+=$jpf-android/../Examples/RSSReader/bin/classes/

#— android setup
android.script=$config_path/InputRSSReader.es
android.checklist = $config_path/CheckRSSReader.cl
android.active_checklists = updateInitiated, getFeedUpdates,
        updateWifiDown, updateBatteryLow, updateRunning
```

Figure 1: Examples of a *.jpf properties file

Checklist definitions are discussed in Section 3.3. They are used to automatically verify that the Android application executes in the expected and correct way. An example Checklist definition file is given in Figure 4. The *.jpf properties file also allows the user to set the active Checklists for this run.

3. THE SCRIPTING ENVIRONMENT

Similarly to how JPF verifies all thread scheduling choices, the scripting environment makes use of JPF's ChoiceGenerator mechanism to schedule the non-deterministic script choices of an ANY script element. This allows the script to contain non-deterministic sequences of input events. When an ANY element is reached in the script, the state of the script is saved and an AlternativeChoiceGenerator schedules each choice non-deterministically for execution.

The input script is grouped into sections each specifying the input events for a specific Window. User events in the script are assumed to be directed to the Window of the current section.

Android applications react to two types of input events: user events and system events.

3.1 User events

User or User Interface (UI) events are triggered by a user interacting with UI elements on the screen or with the physical buttons on the device. This section describes how UI events are scripted and how these events are simulated by JPF-Android.

In Android, all interactive UI elements such as Buttons and TextBoxes are represented by View objects. View objects store the unique id of the element, its state (enabled / visible / focused), its properties (coordinates / size / background / theme) and all of its registered event listeners. Some View objects, such as a LinearLayout and RadioGroup, are capable of containing other View objects. They are called ViewGroup objects. View and ViewGroup objects are stored in a View hierarchy corresponding to a tree structure. The leaves of the tree are View objects and the inner nodes are ViewGroup objects. This View hierarchy is stored inside of a Window object. Each Activity component of the application is associated with a single Window. The Window provides functionality to interact with its View hierarchy. In the Android OS, each Window is assigned a Surface object on which to draw its graphical representation.

JPF-Android closely models the Android UI functionality to ensure that it reacts in the same way as the OS. JPF-Android does this by tracking the state of each View object to avoid triggering actions on disabled or invisible objects from the script. Common View objects such as CheckBoxes, TextBoxes and ListBoxes each store their own unique attributes and listeners. The CheckBox, for example, has a checked attribute and an OnCheckedChangeListener and the ListBox has an adapter to store the list of values to display. The TextBox stores a String value representing the text inside of the box. As Android contains many View components, each with many attributes, these models are continually upgraded and extended. The View hierarchy is stored to support cascading attributes such as visibility.

The script allows the user to simulate UI events, with specific parameters, on View Objects. UI events consists of three parts:

```
UIEvent = "$", target, ".", action, "(", params, ")"
```

target the name of a specific widget at which this action is targeted for example a button, checkbox, textbox.

action the action to perform on the target for example click, enter text, select item in a list.

params optional, comma separated list of parameters.

`$buttonOK.onClick()`, for example, describes a click action on the button `buttonOK` and `$list.selectItem(5)` describes selecting the fifth item in `list`.

This approach requires each View object to be associated with a unique name in its Window. Typically the View's name/ID attribute is used. Views dynamically created from the code do not have a name attribute. These Views are referenced from the script by using a name inferred from the state of the View object. A button for example may use its label text. Physical buttons on the device are not represented by View objects in the hierarchy. To interact with them from the script, the target is set to "device". The device contains reserved actions for simulating interaction with the device. `device.rotate("landscape")` rotates the device to landscape mode and `device.pressBackButton()` simulates pressing the back button on the device.

UI events triggered from the input script are passed to the JPF-Android WindowManager. The WindowManager keeps a reference to the currently visible Window. The View hierarchy of the current Window is then traversed to locate the actual View object. If the View has a registered listener corresponding to the action, this listener is directly triggered on the main thread.

3.2 System Events

Android applications are also driven by system events. System events are fired by the Android system in response to a change in the system state or by other applications interacting with this application. They include notifications such as the state of the WiFi connection changing when the WiFi signal drops.

Although some system events can be triggered manually

```
device.setWifi("OFF");
device.setBattery("LOW");
device.sendSMS("084 123 1234", "Test");
device.setGPS("-33.928806","18.415106");
```

Figure 2: Examples of changing the system state

or by using the Android Debug Bridge (ADB) tool, JPF-Android aims to trigger all types of system events programmatically. This functionality is very important as certain application behavior can only be triggered by system events. We also need to consider that if the system broadcasts a specific state change, the application can query the system for its state. When the system sends a network change event, for example, it is customary for the application to query the current status of the network to make sure it has the newest version of the network state. For the tools running on the emulator this is inherently true. For JPF-Android, however, this results in not allowing events to be sent from the script directly to the application. These events have to be intercepted and forwarded to the relevant service manager to make sure its state also reflects the change.

JPF-Android allows the user to change the state of the system to induce system events. Examples of these state change events are given in Figure 2.

3.3 Runtime verification

Runtime verification validates the correctness of an application by monitoring the execution of the application and comparing it to user defined property specifications. These property specifications can be described using a combination of temporal logic, monitoring operations and regular expressions. Techniques, such as logic based monitoring and error patterns analysis, are then used to verify the application's execution against these specifications [2, 6, 3].

Application monitoring can either be done inline or offline [4]. Inline monitoring requires modifying the application's code to contain annotations/comments describing the property specifications of the program. During a pre-compilation stage, these comments/annotations are used to dynamically generate code verifying these properties. Offline monitoring entails instrumenting the byte-code of the application to emit events indicating changes in the program state. These event traces can then be processed by another system, independent of the application, to detect errors.

JPF-Android makes use of an offline runtime monitoring approach. It employs a JPF Listener to be notified of method invocations. JPF-Android allows methods to be registered as "Checkpoints" in the application code by annotating them with Checkpoint annotations, or by mapping them to Checkpoints in the Checklist definition file. When a Checkpoint method is invoked, JPF-Android perceives this as reaching a specific state in the application code.

JPF-Android allows the user to define a list of Checkpoints, called a Checklist, to verify the flow of the application's execution during runtime. These Checklist's Checkpoints (corresponding to method invocations) are then matched against the application stack-trace to verify the order in which methods are invoked.

Android applications have a single-threaded, asynchronous design. All input events are placed in a message queue, waiting to be handled by the one main thread of the application. However, many applications employ additional threads to perform long running or blocking operations as these operations can cause the main thread to block, leading to a unresponsive graphical user interface (GUI). Due to the GUI of Android applications not being thread safe, these additional threads can only interact with the GUI by posting requests in the main thread's message queue [10]. This makes monitoring Android applications, in terms of method invocations, challenging since methods are invoked on specific threads and multiple threads can be scheduled in different ways.

A Checklist can be defined using the following syntax:

```
name ":" guard "=>" checkpoints ";"
```

A Checklist contains a guard to identify the point in the execution where JPF-Android must start matching its Checkpoints. This is necessary as concurrent applications have multiple threads of execution. These threads can each be verified alone or concurrently. The guard of a Checklist consists of a list of method invocations that have to be reached before the Checklist is marked as active. If the guard is matched in the main thread of execution, the Checklist is checked concurrently for all threads spawning after this point. If the guard is only matched in one of the concurrent threads, only this thread's execution will be verified.

The state of an Android application running on JPF-Android can be changed non-deterministically from the input script using an ANY script element. As a result, an input event can cause different executions of the application for these different application states. The guard also allows the user to specify the specific state of the system for which a Checklist must be verified.

Android applications have a very strong event driven design in which the application execution is prompted by user and system events. To make Checklists more intuitive to this design, a Checklist can only verify the execution prompted by a single input event. Although a Checklist can only verify the execution of an input event, they are defined "Globally" to the entire application execution. In other words, Checklists are fired each time the first Checkpoint in the guard is reached. But, JPF-Android is only interested in Checklists that have matched their guard and have not been completed or have not yet violated a negative Checkpoint.

A Checklist specifies that its Checkpoints have to be reached in a specific order. It is not, however, concerned with which methods are called in between these Checkpoints or whether Checkpoints are reached directly after each other. It simply verifies that when the application is in a specific state and receivers a specific event, that the application follows this specific flow of execution. Checklists also supports negative Checkpoints, which indicate that a Checkpoint may not be reached before the next non-negative checkpoint in the Checklist is reached. A negative Checkpoint is represented by placing a "!" symbol in front of the Checkpoint

Checklists are designed to be succinct logical execution spec-

```
1. SECTION default {
2.   @urlInputStreamIntent.putString("url","http://feed.rss")
3.   @urlInputStreamIntent.putString("file","src/input.rss")
4.   sendBroadcast(@urlInputStreamIntent)
5.   @startIntent.setComponent("TimelineActivity")
6.   startActivity(@startIntent)
7. }
8. SECTION TimelineActivity {
9.   ANY{
10.     GROUP{
11.       device.setWifi("ON")
12.       device.setBattery("100%")
13.     },
14.     GROUP {
15.       device.setWifi("OFF")
16.       device.setBattery("100%")
17.     },
18.     GROUP {
19.       device.setWifi("ON")
20.       device.setBattery("LOW")
21.     }
22.   }
23.   $buttonUpdate.click()
24. }
```

Figure 3: Input Script

```
updateInitiated: update, checkRunning, !alreadyRunning,
        checkWifi, !WifiDown, checkBattery, !batteryLow
        => runUpdate;

getFeedUpdates : parseFeed
        => storeInDB, loadFromDB, updateListView;

updateWifiDown: update, checkWifi, WifiDown
        => !runUpdate, cancelUpdate, notifyWifiDown;

updateBatteryLow: update, checkBattery , batteryLow
        => !runUpdate, cancelUpdate, notifyBatteryLow;

updateRunning: update, checkRunning , alreadyRunning
        => !runUpdate, cancelUpdate, notifyAlreadyRunning;
```

Figure 4: Checklist Definitions

ifications. Their aim is not to detect safety and liveness properties, but to automatically verify the application execution during runtime.

4. EVALUATION

The evaluation section looks at how JPF-Android can be used to verify properties of a RSSFeedReader application. The application downloads and stores RSS feed items in a database. The most current items are then displayed in a list combining items of all the RSS feeds, ordered by creation time. When an item in the list is selected, a web page is shown containing the item's content.

This example illustrates how JPF-Android is capable of modeling a network connection, XML parser, a database connection and also three of Android's main components namely an Activity, Service and Broadcast Receiver.

JPF-Android aims to provide the user with as much freedom as possible to configure external input to the application. This input can be non-deterministically changed by using an ANY script element. The network connection is modeled by allowing the user to associate a URL with a filename containing the data sent over the network from within the input script. In Figure 3, line 2-4 a specific predefined Intent object, called a `urlInputStreamIntent`, is constructed and sent to the system from the script. When a network connection is made to this URL, JPF-Android returns the specified file's contents. This input stream is then sent to the XML parser to be parsed.

When the user clicks on the update button in the TimelineActivity, the application must retrieve the newest items from each registered feed and update the list to reflect these changes. For simplicity's sake we only allow updates over an active WiFi connection. If the user clicks the refresh button, and the WiFi is not connected or the battery is too low or a update is already running, the application must not attempt to update the feeds but notify the user. To verify that the application correctly handles each of these situations, we use the input script in Figure 3 to simulate each of these situations non-deterministically and then register the Checklists in Figure 4 to verify this behavior.

Checklist can be violated in two ways after the guard has been matched: firstly, when a negative Checkpoint in the Checklist is reached or when all Checkpoints in a Checklist have not been reached in order, before the search has ended. Checklists are matched during runtime and violating Checklists are reported at the end of execution. To illustrate how Checklists are reported, we have introduced two violations into the RSSFeedReader application. Figures 5 and 6 displays these violations as reported by JPF-Android. In Figure 5 the "getFeedUpdates" Checklist failed due to the "storeInDB" Checkpoint not being reached. In Figure 6 the application tried to update although the WiFi was off, which violated the "!runUpdate" Checkpoint in the "updateWifiDown" Checklist.

5. RELATED WORK

The main challenge of testing Android applications is generating and simulating input events. This section looks at how other projects approached these challenges.

UI Automator[2] allows the user to script any manual action the user would normally be able to perform on the device and then execute these events, one by one, on the emulator/device. But, some system events like battery low notifications, triggering an alarm or testing the behavior of the application using specific input parameters are not as easily triggered using this approach.

Android's monkey tool is a stress testing tool that takes a black-box approach to Android application testing[3]. It automatically generates pseudo-random input events and fires the events while the application is running on the emulator / device. Input events can be generated quite quickly as they are randomly selected, but it can also generate many unimportant event sequences and repeat unimportant events while not triggering important events. MonkeyRunner[4] provides a python API for users to script input event sequences that are simulated by the Android emulator. Dynodroid

[2] `developer.android.com/tools/help/uiautomator`
[3] `developer.android.com/tools/help/monkey.html`
[4] `developer.android.com/tools/help/monkeyrunner_concepts.html`

```
=================================== violations
Checklist Name: getFeedUpdates
EventID: 8
Reason: Checkpoint storeInDB not visited before
        the search ended.
Script Events:
1.  @urlInputStreamIntent.putExtraString("url"...
2.  @urlInputStreamIntent.putExtraString("file"...
3.  sendBroadcast(@urlInputStreamIntent)
4.  @startIntent.setComponent("TimelineActivity")
5.  startActivity(@startIntent)
6.  device.setWifi(``ON'')
7.  device.setBattery(``100%'')
8.  $buttonUpdate.click()
=================================== results
```

Figure 5: Violation 1

```
=================================== violations
Checklist Name: updateWifiDown
EventID: 8
Reason: Failed because checkpoint reached runUpdate
    did not match checkpoint !runUpdate.
Script Events:
1.  @urlInputStreamIntent.putExtraString("url",...
2.  @urlInputStreamIntent.putExtraString("file",...
3.  sendBroadcast(@urlInputStreamIntent)
4.  @startIntent.setComponent("TimelineActivity")
5.  startActivity(@startIntent)
6.  device.setWifi(``OFF'')
7.  device.setBattery(``100%'')
8.  $buttonUpdate.click()
=================================== results
```

Figure 6: Violation 2

makes use of the same approach as MonkeyRunner where it sends input events to the OS using the ADB daemon running on the device to fire UI and system events [7]. Its main approach it not to let the user script these input events, but, automatically choosing them. Before choosing an event it firstly observes all events that influences the state of the application. It then intelligently selects one of the events, careful not to starve certain event types and executes the event on the device.

Another way to test Android application execution is using Android's JUnit testing framework for Android[5]. This framework allows the developer to write unit and integration tests for Android applications. It supports Android component testing and supports testing the interaction between Activities and Services. The disadvantage of this framework is that the state of the emulator can not be changed while the tests are run. This means that we can not programmatically send system events to change the battery state or the WiFi radio's state of the device. Robotium is a project that extends the Android JUnit framework and improves its syntax[6], but it still includes the same limitations as the JUnit framework. Model-based testing approaches extract models of the behavior of a system and then utilize these models to generate correct and meaningful JUnit test cases and input values to test the system. Various projects [11, 1] have developed solutions to extract a model from an Android application and use this model to generate test cases that can run on the emulator.

Robolectric is also a JUnit testing framework for Android, but runs outside of the emulator in the JVM. As the Android source code can not compile on the JVM, Robolectric makes use of the JavaAssist library to intercept class loading and return a shadow class that support the same type of functionality as the original class [9].

Other projects focus on applying symbolic verification techniques such as concolic testing [5] or symbolic execution [8] to automatically generate valid inputs while traversing the application.

6. CONCLUSION

In the paper we presented an approach to verify Android applications on the Java Pathfinder JVM. Many of the other available testing frameworks for Android verify applications on an emulator/device which has limited resources and limited exposed functionality. JPF-Android verifies all execution paths of an Android application using non-determinism to explore all possible thread interleavings and many different input event sequences. The paper also presents a runtime verification framework to automatically verify that the Android application executes in the expected and correct way.

The largest limitation of JPF-Android is the size and complexity of the input script. As JPF-Android uses model-checking to verify application properties, it suffers from the state explosion problem. This limits the length and complexity of the input script.

Our future work includes extending Checklists to support choices (OR operator) and running a large example application on the system to analyze the size of the state space the tool generates.

[5] developer.android.com/tools/testing

[6] code.google.com/p/robotium

7. REFERENCES

[1] D. Amalfitano, A. R. Fasolino, and P. Tramontana. A GUI Crawling-Based Technique for Android Mobile Application Testing. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 252–261. IEEE, Mar. 2011.

[2] F. Chen and G. Roşu. Towards Monitoring-Oriented Programming. *Electronic Notes in Theoretical Computer Science*, 89(2):108–127, Oct. 2003.

[3] K. Havelund and G. Roşu. Monitoring Java Programs with Java PathExplorer. *Electronic Notes in Theoretical Computer Science*, 55(2):200–217, Oct. 2001.

[4] K. Havelund and G. Roşu. Efficient monitoring of safety properties. *International Journal on Software Tools for Technology Transfer*, 6(2):158–173, Nov. 2003.

[5] C. S. Jensen, M. R. Prasad, and A. Møller. Automated Testing with Targeted Event Sequence Generation. In *Proc. 22nd International Symposium on Software Testing and Analysis (ISSTA)*, Lugano, Switzerland, July 2013.

[6] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Java-MaC: a Run-time Assurance Tool for Java Programs. *Electronic Notes in Theoretical Computer Science*, 55(2):218–235, Oct. 2001.

[7] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An Input Generation System for Android Apps. *ACM SIGSOFT Software Engineering Notes*, Aug. 2013.

[8] N. Mirzaei, S. Malek, C. S. Pasareanu, N. Esfahani, and R. Mahmood. Testing android apps through symbolic execution. *ACM SIGSOFT Software Engineering Notes*, 37(6):1, Nov. 2012.

[9] B. Sadeh and S. Gopalakrishnan. A Study on the Evaluation of Unit Testing for Android Systems. *International Journal of New Computer Architectures and their Applications (IJNCAA)*, 4(1):926–941, 2011.

[10] H. van der Merwe, B. van der Merwe, and W. Visser. Verifying Android applications using Java PathFinder. *ACM SIGSOFT Software Engineering Notes*, 37(6):1, Nov. 2012.

[11] W. Yang, M. R. Prasad, and T. Xie. A Grey-Box Approach for Automated GUI-Model Generation of Mobile Applications. In *Fundamental Approaches to Software Engineering*, volume 7793 of *Lecture Notes in Computer Science*, pages 250–265. Springer Berlin Heidelberg, 2013.