Verifying Android Applications using Java PathFinder

Heila van der Merwe, Brink van der Merwe and Willem Visser
Dept. of Computer Science
University of Stellenbosch
Private Bag X1 Matieland
South Africa, 7602
hvdmerwe@cs.sun.ac.za, abvdm@cs.sun.ac.za, wvisser@cs.sun.ac.za

ABSTRACT

Mobile application testing is a specialised and complex field. Due to mobile applications' event driven design and mobile runtime environment, there currently exist only a small number of tools to verify these applications.

This paper describes the development of JPF-ANDROID, an Android application verification tool. JPF-ANDROID is built on Java PathFinder, a Java model checking engine. JPF-ANDROID provides a simplified model of the Android framework on which an Android application can run. It then allows the user to script input events to drive the application flow. JPF-ANDROID provides a way to detect common property violations such as deadlocks and runtime exceptions in Android applications.

Categories and Subject Descriptors

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

General Terms

VERIFICATION

Keywords

Mobile Application, Java PathFinder, JPF, Android, Verification, Testing

1. INTRODUCTION

Software testing and verification plays an important role in determining the quality and robustness of software. These are two very important attributes of mobile applications. Especially since users currently have more than 600 000 applications to choose from on the Google Play Market.

Although software testing is so important, it is often neglected due to its complex and time consuming process. Android applications face additional challenges when it comes to application testing. Firstly, they have an event based design which means that their application flow is driven by graphical user interface (GUI) and system events. Secondly, Android applications are developed in a custom implementation of the Java Application Programming Interface (API) adopted from the Apache Harmony [10] project. The compiled applications can only be executed on a special virtual machine (VM) called the Dalvik VM that runs on Android devices [5].

Due to these challenges, the rapid pace at which mobile applications are developed and the lack of testing tools available, mobile application testing is often omitted altogether. The simplest way to test GUI applications is manual black-box testing. But, this is a time consuming, error prone and expensive [6] process. One way to reduce this high cost, is to automate the testing of GUI – in this case Android – applications [6].

The most common way to automate Android application testing is to run tests on the Dalvik VM on a physical device/emulator. Testing frameworks such as the MonkeyRunner and Robotium make use of Android's built-in JUnit framework [4] to run test suites on the device. Running tests on the Dalvik VM is slow as they have to be instrumented and each test sequence has to be defined and executed sequentially. Other projects use alternative ways to automatically generate input by manipulating the built in accessibility technologies [6] or by re-implementing the Android keyboard [8]. Mockito and Android Mock allow JUnit testing on the Dalvik VM by using mock Android classes. The advantage of testing applications on the Dalvik VM is that we can physically emulate input events on the device. The disadvantage is that it is not simple to automate this input emulation.

Another approach is to test Android applications using the Java Virtual Machine (JVM). There are different ways of implementing such a framework. Android lint, for example, makes use of static analysis to identify common errors in applications. Robolectric [3], a JUnit testing framework running on the JVM, intercepts the loading of Android classes and then uses shadows classes that model these classes.

Although Android applications and Java desktop applications are designed for completely different Java VMs, they are both built on an implementation of the Java API. As a consequence, Android applications contain many of the

same error as Java applications. These defects include, but are not limited to, concurrency issues and common runtime exceptions such as NullPointer Exceptions. It follows that existing Java testing frameworks can be adapted to verify Android applications.

This paper describes an extension to Java PathFinder (JPF) [2] that enables the automatic verification of Android applications on the standard Java JVM.

The next section will provide an overview of JPF and Android and how JPF can be used to test Android applications. Thereafter the development of the JPF-ANDROID tool will be discussed.

2. DESIGN

2.1 Java PathFinder

JPF is an automated, open source, analysis engine for Java applications [2]. It is implemented as an explicit state model checker that includes mechanisms to model Java classes and native method calls (Model Java Interface - MJI Environment), track byte code execution and listen for property violations. Additionally, JPF's design encourages developers to create extensions to the framework. Currently there exist many extensions including a symbolic execution extension (JPF-SYMBC), data race detector (JPF-RACEFINDER) and an abstract window toolkit (AWT) extension (JPF-AWT) [9].

JPF-AWT allows the model checking of AWT applications. AWT applications are event driven and based on a single threaded, message queue design. All application events are put in a message queue and then handled by the main thread of the application, called the EventDispatchThread. JPF-AWT introduced the idea of using a simple event script file to write sequences of user inputs to drive the application execution. JPF-AWT models the EventDispatchThread so that when the message queue is empty, it requests an event from the script file simulating the event occurring.

JPF-ANDROID makes use of JPF's extension mechanisms to model the Android application framework so that Android applications can run on the JVM. It then extends JPF-AWT's input model to simulate user and system input to drive the application flow. One of the advantages of extending JPF is that it has been rigorously tested and can successfully detect many common defects in Java applications. As soon as Android applications can run on JPF-ANDROID, common software errors are automatically detected.

2.2 Android

Android is an open source software stack for devices with an advanced RISC Machine (ARM) architecture. It consists of the Android operating system (OS), the application framework and an application development toolkit assisting developers to create applications for the platform [1].

As shown in figure 1, the Android OS is built on top of a modified Linux kernel. The kernel provides a layer of abstraction on top of its low level functionality such as process, memory, user, network and thread management. On top of

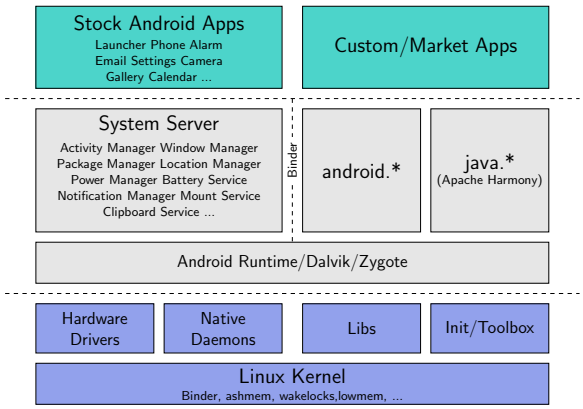

Figure 1: The Android application stack [13]

the Android kernel is a set of native C-libraries. The Dalvik VM is a custom, optimised version of the JVM. For security reasons each Android application is run as a separate process in its own Dalvik VM instance [12]. Hence, applications can only communicate with each other and the with the Android framework using Android's Binder inter-process communication (IPC) mechanism [11].

Android has one main application called the system process. The system process contains services responsible for performing the main tasks of the system including:

ActivityManager manages the life-cycle and interaction of all the activities running on the system

WindowManager allows applications to draw on the screen and forwards UI input to the application.

PackageManager stores information on the application packages installed on the device

All Android applications follow a single-threaded design in which the main thread of the application handles all application events [1]. This structure is commonly used by many UI frameworks since it becomes too complex to make all UI classes thread safe [7]. In Android, this main thread is called the `Looper` thread. The `Looper` has a message queue attached to it containing all application events to be dispatched. UI and system events are scheduled on the main `Looper` by adding them to the message queue. The `Looper` is responsible for continuously looping through these messages and handling them appropriately. This could include updating a widget, loading a new activity or processing an Intent.

The main entry-point of each Android application is in its `ActivityThread` class. The `ActivityThread` class is part of the Android framework and starts the application's `Looper` thread. It also keeps track of the application's components and handles user and system events. Android applications also consists of the following application components:

Activity responsible for representing and managing an user interface. An application can consist of many Activities.

Service performs background operations such as the pulling of messages from a server every 5 minutes. Services do not have user interfaces and an application can have zero or more services running simultaneously.

Broadcast receiver listens for and responds to system-wide events such as network failing, low battery or screen orientation change events.

Content provider manages application data stored on the file system, in databases, on the web or other storage medium and provides a gateway to this data from other applications.

These components interact with each other and with other applications using a structure called an `Intent`. An `Intent` is a high level implementation of the Binder IPC. `Intents` are used to start Activities and Services or to provide a notification of certain events. They are similar to messages containing a description of an operation to be performed or, often in the case of broadcasts, a description of something that has happened and is being announced [1].

2.3 Scope of JPF-ANDROID

The objective of JPF-ANDROID is to verify Android applications by running the application code on the JVM using a collection of different event sequences and then to detect when certain errors occur using JPF.

The Android framework is very large and one of the main challenges of JPF-ANDROID is to decide which parts of the system to model. The more of the Android framework is modelled, the more realistic the model is and the more errors can be found. But, if too much of the framework is modelled the scheduling possibilities increase exponentially which means that the search space can become too big to verify.

JPF-ANDROID focuses on verifying a single application with multiple application components and their interaction. The system service of the Android OS is not part of the application process and runs in its own thread. To reduce scheduling possibilities, the entire system service is not modelled but its necessary components are implemented as part of the application process. The following parts of the Android framework is modelled:

ActivityManager manages the life-cycle of Activities and other application components.

ActivityThread the main entry-point to the application. It controls and manages the application components and their input.

Application components including Activity, Service, Broadcast Receiver and Content Provider.

Window and View structure The view hierarchy is modelled including the widgets and the window classes.

Message queue modelled to support script input.

Lastly, as the application will not be communicating with outside processes, the `Intent` objects are modelled to exclude the Binder IPC service.

```
1. @intent1.setComponent("SampleActivity")
2. startActivity(@intent1)
3. $button1.onClick()
```

Figure 2: Starting SampleActivity

3. DEVELOPMENT

3.1 JPF-ANDROID architecture

As discussed above, both Android and AWT applications have a single-threaded application design. That is why JPF-ANDROID's architecture is based on JPF-AWT. JPF-ANDROID models Android's message queue by using JPF's MJI environment. When the `Looper` thread requests a new message from the message queue and it is empty, a call to the native `JPF_MessageQueue` class requests the next event from the scripting environment. The `JPF_MessageQueue` class classifies events as either UI or system events. UI events are directed to and handled by the native `JPF_Window` class and system events by the native `JPF_ActivityManager` class.

UI events include events fired by widgets such as clicking a button or selecting an item in a list view. An Activity's UI is represented by a window object containing a view hierarchy. When the window is inflated, an object map is created in the native `JPF_Window` class. This map binds the name of each widget to the reference of the inflated widget object. When an UI event is received by the native `JPF_Window` class, the name of the target widget is looked-up in the object map and the action is then called directly on the inflated widget object by pushing a direct call frame on the JPF call stack.

System events use `Intent` objects to describe a the event that occurred. They include events to start an Activity or Service or a battery low notification. In the Android OS, system events are sent to the Activity manager which is modelled in JPF-ANDROID by the native `JPF_ActivityManager` class. This class keeps a map of Intents as they are defined in the script file. When a system event is received, the corresponding `Intent` is looked up in the map. The `Intent` is then resolved to the relevant component and scheduled in the application's message queue.

3.2 The scripting environment

JPF-AWT's script consist of a list of UI events. These events are executed one-by-one when the message queue it empty. To allow users to script system events, variables were added to the scripting language in JPF-ANDROID. Variable names are identified by the "@" in front of the variable's name as '$' is already used in JPF-AWT to identify UI components. These variable can be used to construct most Intent objects. For example, a script file that sends an `Intent` to start the `SampleActivity` is shown in figure 2

JPF-ANDROID scripts also adopted the **REPEAT** and **ANY** constructs from JPF-AWT. **REPEAT** constructs repeat a list of events a specified number of times. **ANY** constructs take a list of events as parameters and then uses a ChoiceGenerator and JPF's state matching and backtracking features to visit each of the execution branches. JPF stores the state of the system before the it advances to a new state. JPF-AWT uses a JPF search listener to store

```
1. @startIntent.setComponent("ListContactsActivity")
2. startActivity(@startIntent)
3. $createContactButton.onClick()
4. $nameEdit.setText("Mary")
5. ANY { $<create | clear>Button.onClick()}
6. $nameEdit.setText("Maria")
```

Figure 3: Original script for Contacts application

```
1. SECTION default {
2.     @startIntent.setComponent("ListContactsActivity")
3.     startActivity(@startIntent)
4. }
5.
6. SECTION ListContactsActivity {
7.     $createButton.onClick()
8.     $list.setSelectedIndex([0-3])
9. }
10.
11. SECTION NewContactActivity {
12.     $nameEdit.setText("Mary")
13.     ANY { $<create | clear>Button.onClick()}
14.     $nameEdit.setText("Maria")
15. }
```

Figure 4: Adding sections to the input script

and retrieve the current position of the input script in a specific state so that the script's state is also saved.

Android applications contain multiple windows - one for each Activity. This complicates the script input as each window has its own unique set of view components, hence, a unique input sequence. Furthermore, the application flow can switch between these Activities at any point of execution. In other words, if we do not know in what Activity the application is, we can not script the following events.

Let us take a contacts android application as an example. The application has two Activities: ListContactsActivity and the NewContactActivity. The ListContactsActivity contains a list of the current contacts and a button to add new contacts that directs the user to the NewContactActivity. The NewContactActivity contains two buttons: the clear button stays on the current Activity and clears its fields and the create button starts the `ListContactsActivity`. Now let us look at figure 3 containing the input sequence for the NewContactActivity.

The problem occurs in the ANY structure. When the create button is clicked the application changes to the ListContactsActivity and the following event is not valid any more. This drastically restricts the number of sequences that can be scripted in the file. To address this issue, we included the use of sections in the input script (see figure 4). Each section groups the input events of a specific Activity. Now, if the create button is pressed the `ListContactsActivity` will be started and the `ListContactsActivity`'s events specified in its sequence will start executing again.

The addition of the sections in the input script lead to in-

```
1. SECTION SimpleActivity {
2.     ANY { $button[0-9].onClick() }
3.     ANY { $button<Plus|Minus|Mul|Div|More>.onClick() }
4.     ANY { $button[0-9].onClick() }
5.     $buttonEquals.onClick()
6. }
7. SECTION ScientificActivity {
8.     ANY { $button<Sin|Cos|Tan>.onClick() }
9. }
```

Figure 5: Input script for Calculator application

finite event sequences. In the above example, an infinite loop occurs when the create button is pressed on the NewContactActivity window. This will stop the execution of the NewContactActivity section and restart the ListContactsActivity section's events. When the ListContactsActivity's section is restarted, it again clicks on the createButton restarting the NewContactActivity. To address this issue, JPF-ANDROID keeps record of its current Activity. The scripting environment was also adapted to store the current position of each section's events for each visited Activity. If the branch returns to a previously visited Activity, its current position in its section is looked up and instead of restarting the section, it continues from its previous position.

3.3 Case studies

The first application is a scientific calculator. The calculator has two Activities: a simple view that displays basic arithmetic operations and a scientific view that displays more complex arithmetic operations. When the user switches between these Activities, the current state of the calculator is preserved. This state includes the intermediate values and current operation. This state information is bundled with the `Intent` that starts then next Activity.

The calculator application contains two errors. When the user divides a value by zero, an uncaught `ArithmeticException` is thrown by the application which causes Android to kill the application. Secondly the application neglected to attach the state information to the Intent that is passed to the next Activity. When the user switches to the other Activity a `NullPointerException` is thrown when it tries to read the state information from the Intent.

To detect these errors we will use the test script in figure 5. The scripting environment interprets the script into the following input sequences:

$$\begin{aligned}
\text{<sequence>} &= \text{<simple_seq>} \mid \text{<complex_seq>} \\
\text{<simple_seq>} &= \text{number, simple_op, number, equals} \\
\text{<complex_seq>} &= \text{number, complex_op} \\
\text{simple_op} &= \text{“+”} \mid \text{“-”} \mid \text{“} \times \text{”} \mid \text{“} \div \text{”} \\
\text{complex_op} &= \text{“sin”} \mid \text{“cos”} \mid \text{“tan”} \\
\text{equals} &= \text{“=”} \\
\text{number} &= \text{“0”} \mid \text{“1”} \mid \text{“2”} \mid \text{“3”} \mid \text{“4”} \mid \text{“5”} \mid \text{“6”} \mid \text{“7”} \mid \text{“8”} \mid \text{“9”}
\end{aligned}$$

Each of the ten number buttons are pressed followed by a simple operation and another number or a complex operation.

```
1. SECTION DeadlockActivity {
2.     $button1.onClick()
3.     $button2.onClick()
4. }
```

Figure 6: Input script for Deadlock application

JPF is automatically configured to detect thrown exceptions and to stop execution. When JPF-ANDROID was run it firstly detected the `ArithmenticException` due to the division by zero:

```
====================================================== results
error #1: gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty
        "java.lang.ArithmeticException: Division by zero  at ..."
====================================================== statistics
elapsed time:       00:00:01
states:             new=76, visited=0, backtracked=63, end=30
search:             maxDepth=13, constraints hit=0
choice generators:  thread=7 (signal=0, lock=3, shared ref=0), data=39
heap:               new=3455, released=2105, max live=1507, gc-cycles=74
instructions:       76796
max memory:         117MB
loaded code:        classes=145, methods=2064
======================================================
```

After this error was fixed JPF-ANDROID detected the `NullPointerException` in the same way.

Both of these errors would have been difficult to detect with unit testing. The `ArithmenticException` is challenging due to the many possible input sequence combinations. If a test case did not specifically identify this as a point of interest, unit testing would not have detected this error. The second error is challenging to detect due to the fact that it only occurs when the flow of Activity classes are modelled.

The next case study is a very simple application demonstrating how JPF-ANDROID detects a deadlock in a Android application. When the `Looper`/main thread of an application is caught in a deadlock, the Android OS kills the application and displays an Application Not Responding (ANR) dialog. But, Android does not detect a deadlock if it occurs between other asynchronous threads. This sample application spawns two asynchronous threads that deadlock. The application has one Activity with two buttons. The first button spawns the first thread and the second button spawns the second thread. After a while these two thread deadlock and are then blocked forever, waiting for each other. The input script for the application is given in figure 6.

JPF is then configured to listen for deadlocks and schedules the threads in all possible ways to detect the deadlock.

```
 ====================================================== thread ops #1
   1       1     trans          loc             : stmt
------- ------- ------------------------------------------------------
B:1003     |      54  DeadlockActivity.java:82 : bower.bowBack(this);
   |    B:1000    54  DeadlockActivity.java:82 : bower.bowBack(this);
L:1000     |      54  DeadlockActivity.java:56 : friend[0].bow(friend[1]);
   |    L:1003    18  DeadlockActivity.java:56 : friend[0].bow(friend[1]);
   S       |       6
   |       S       3
====================================================== results
error #1: gov.nasa.jpf.jvm.NotDeadlockedProperty
        "deadlock encountered:   thread java.lang.Thread:{i..."
====================================================== statistics
elapsed time:       00:00:01
states:             new=55, visited=13, backtracked=67, end=1
search:             maxDepth=10, constraints hit=0
choice generators:  thread=26 (signal=0, lock=11, shared ref=0), data=7
heap:               new=1430, released=375, max live=1040, gc-cycles=67
instructions:       12661
max memory:         117MB
loaded code:        classes=140, methods=1792
======================================================
```

4. FUTURE WORK

Currently JPF-ANDROID can detect deadlocks, race conditions and other property violations in Android applications. The next challenge is modelling the extra Android libraries. This is because most Android applications make use of many Android specific libraries such as the sqlite database connector, HTTP connections or the media player libraries.

Another extension that will be added to JPF-ANDROID is coverage testing. JPF has many coverage extensions available so we will be adapting one these extensions to work on Android applications.

5. CONCLUSION

The paper discussed the design and implementation of JPF-ANDROID. JPF-ANDROID is still under development and currently only models the core libraries needed to verify a basic Android application. It allows Android applications to be tested using JPF's proven verification techniques and can successfully detect common Java errors such as runtime exceptions and deadlocks.

This extension provides a basis on which Android applications can be tested. It can later be extended to verify functional requirements and identify Android specific errors using JPF's listener mechanism.

6. REFERENCES

[1] Android documentation. `http://developer.android.com/`. Accessed: 17 July 2012.

[2] Java Pathfinder documentation. `http://babelfish.arc.nasa.gov/trac/jpf`. Accessed: July 2012.

[3] Robolectric documentation, November 2007. `http://pivotal.github.com/robolectric`. Accessed: July 2012.

[4] Testing fundamentals, June 2012. `http://developer.android.com/tools/testing/`. Accessed: 17 July 2012.

[5] D. Ehringer. The Dalvik virtual machine architecture. 2010.

[6] M. Grechanik, Q. Xie, and C. Fu. Creating gui testing tools using accessibility technologies. In *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 243 –250, April 2009.

[7] G. Hamilton. Multithreaded toolkits: A failed dream?, October 2004. `http://weblogs.java.net/blog/kgh/archive/2004/10/multithreaded_t.html`. Accessed: 17 July 2012.

[8] H. Ji. Mobile software testing based on simulation keyboard. In Q. Luo, editor, *Advances in Wireless Networks and Information Systems*, volume 72 of *Lecture Notes in Electrical Engineering*, pages 555–561. Springer Berlin Heidelberg, 2010.

[9] P. Mehlitz, O. Tkachuk, and M. Ujma. Jpf-awt: Model checking gui applications. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 584 –587, November 2011.

[10] E. Pasko. Google android, apache harmony and java packaging, 2007. `http://apache-harmony.blogspot.com/2007/11/google-android-apache-harmony-and-java.html`. Accessed: July 2012.

[11] T. Schreiber, J. Somorovsky, and D. Bußmeyer. Android Binder. [seminarthesis], October 2011.

[12] J. Six. *Application Security for the Android Platform*. O'Reilly Media, Inc., December 2011.

[13] K. Yaghmour. Understanding the Android System Server. In *AnDevCon, Android conference*, 2011.