Model checking Android Applications on the Java Virtual Machine

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, abvdm, wvisser}@cs.sun.ac.za

ABSTRACT

JPF-Android is a model checking tool that verifies Android applications outside of the emulator on Java PathFinder (JPF). JPF-Android takes as input a script containing sequences of user input events as well as system input events. It then traverses all execution branches of the application by firing these events on a model of the Android Software stack. This paper describes the design of JPF-Android scripting environment and introduces Checklists to allow the user to automatically verify the flow of execution from within the script.

Categories and Subject Descriptors

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

General Terms

Verification

Keywords

Android application, Java Pathfinder, Testing, Verification, Model Checking

1. INTRODUCTION

Android applications have a distinctly different design to desktop and web applications. Android applications are designed to be used on a mobile platform where multiple applications are used multiple times per day, for a brief period of time. Android facilitates this design by not immediately killing applications that are closed by the user, but keeps as many applications active, in memory and ready to be launched the next time the user re-opens the application [13].

The Android operating system has a hard limit on available memory as it has no swap space available like desktop systems. This limits the number of applications that can be kept in memory at a specific time. To cope with this hard limit, Android keeps a stack of recently used applications. When the memory usage it high, applications are killed from the bottom of this stack to reclaim memory [13].

Android applications have to be very robust, consistent in their behavior and always retain their application state to provide a good user experience. Many verification and testing techniques are being researched to develop a framework that assists developers in creating robust and error free Android applications. Each approach has its own strengths and weaknesses. Some tools, such as Android Lint [3], makes use of a static analysis approach to identify possible errors and security issues in applications. Other tools take a more dynamic testing approach [2, 11, 12, 16]. JPF-Android makes use of a dynamic testing approach to identify errors and property violations by model-checking Android applications [19].

Testing Android applications dynamically provides some interesting research challenges. The most common way to test Android applications is running them on the emulator or an Android device that supports all Android application functionality. However, executing tests in this way is very slow as they have to be compiled, packaged and then uploaded to the device. This process is time consuming and is not a suitable testing approach, especially when using a test driven development approach where the test execution frequency is very high [18]. Unfortunately, Android applications can not directly run on the Java Virtual Machine as they are dependent on the Android SDK including Android operating system services and the application framework that implements the core functionality of an application. The Android SDK code can not directly run on Intel computers as they do not have the same architecture and do not include the same low level system drivers and libraries as ARM devices. As a result Android applications can only be run on the emulator which mimics all of the hardware and software capabilities of a mobile device [1].

Android application are event driven. Testing event driven applications can be intricate as we have to simulate input events to drive the application flow. When combining these events into event sequences, we can end up with millions of different combinations of events, which can all potentially trigger different behaviors of the application. A successfully testing tool must be able to simulate all types of input events including user and system events.

Lastly, Android applications consist of many application components such of Activities, Services and Broadcast Receivers. These components communicate with each other and with the operating system using an asynchronous message passing approach. Due to this asynchronous communication it is difficult to verify the state of the application. Especially when we want to verify that non-GUI components such as Services received the correct input and executed as expected.

The goal of this project is to create a testing tool for Android applications that:

- provides an environment outside of the emulator wherein an Android application can be tested,
- provides a way to input event sequences containing all types of events to drive the application execution,
- allows us to automatically verify specific application properties and application flow.

We achieved these goals by creating a Android application verification tool called JPF-Android. This paper follows from our previous paper "Verifying Android Applications using Java PathFinder" [19], where we describe the design decisions and implementation of the tool. In this paper we focus on the architecture and design of the scripting environment and an extension we have made to verify the execution sequence of Android applications. Lastly we illustrate how our tool can be used to verify the business logic of a RSSFeedReader Android application.

2. SCRIPTING FRAMEWORK

JPF-Android makes use of an input script containing the 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, the next event is requested from the scripting environment.

The script supports "ANY"- structures that split the current event sequence into multiple sequences that are executed non-deterministically. At such a point, the state of the scripting environment is saved, together with the state of the Android application. These sequences are then traversed one by one. After each sequence, the entire application state is backtracked to the point at which the sequence was split before the next sequence is continued.

Android applications rely on two types of input events to drive the application: user events and system events. The next sections of the paper will discuss both of these types of events and how JPF-Android supports these input events.

2.1 User events

User 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 handled by Android and how JPF-Android simulates user inputs.

2.1.1 Android User Interface (UI) design

In Android, all interactive UI elements such as buttons and text boxes are called View objects. View objects store the id of the element, its state (enable / visible / focused), its properties (coordinates / size / background / theme) and all its registered listeners. Some UI elements can contain other UI elements, such as a LinearLayout and RadioGroup. 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 the functionality for the Activity to interact with the View hierarchy. In the Android Operating System, each Window is assigned a Surface object on which to draw its graphical representation.

There are 3 types of UI Events in Android: MotionEvent, KeyEvent and DragEvent.

A MotionEvent is fired when the user touches the screen. It can trigger one of two listeners of a View object: onClickListener or the onTouchListener. A onTouchListerner is more specific than an onClickListener as it specifies the View that was touched and also the location of the touch. If a View has a registered onTouchListener and onClickListener, the onTouchListener will always be the only listener that will trigger for the event. A motion event is handed to the currently visible Activity by the InputManager. The Activity then forwards the event to its Window which passes the event to its View hierarchy. If the event is not handled by any of the Views in the hierarchy, it is passed back to the Activity to be handled here. In other words event are passed through an ordered list of listeners in the system until the event is handled by one of them.

There are two type of KeyEvents. Events fired when the user presses on one of the physical buttons on the device and events fired when the user presses one of the keys on the soft keyboard. When a physical button is pressed a onKeyListener is triggered by the event. Soft keyboard events each trigger a onKeyDown and onKeyUp event per key press. When the user types a key on the keyboard, an event is dispatched from the InputManager to the WindowManager. The WindowManager gets the first opportunity to respond to the event. Here we handle events that are application independent such as the home button being pressed. From here the event is handled the same as MotionEvents.

DragEvents are dispatched when a user drags their finger over the screen. As they are not currently handled by JPF-Android, we will not discuss them here.

2.1.2 JPF-Android's UI design

To ensure that JPF-Android can model most of the Android UI functionality while simplifying its implementation, JPF-Android was implemented in the following way.

Firstly, JPF-Android tracks the state of each View to avoid triggering actions on disabled or invisible Views from the script. These two attributes are implemented as boolean values in the View class. Currently JPF-Android is modeling a very basic version of focused state of a View, where we assume focus is given to a view when an action is performed on the widget from the script. The state of specific View objects such as a CheckBox, TextBoxes and ListBoxes are also modeled. The CheckBox, for example, has a checked attribute 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. All further graphical functionality and properties are stubbed out and mocked as we will not physically draw a Window on the screen and we do not support verifying these properties.

The View hierarchy is stored to support cascading state properties such as visibility. This is also useful when traversing the view hierarchy from within the application code. Lastly, each View will store its own listeners that can be triggered by specific actions. Just like in Android, each View contains the base listeners for Motion and Key Events. But some views such as the ListView and CheckBox also contain custom listeners for their specific attributes.

From the script, the user is able to trigger an action with specific parameters on a View to simulate interaction with

the UI. UI Events are scripted using the following syntax:
$<VIEW>.<ACTION>[< *ARGUMENTS* >]

An example of this would be simulating a button click or by using &button1.onClick() or selecting an item in a list by using &updatesList.selectItem(5).

This approach requires that each widget must be associated with a unique name in its Window. The view hierarchy of a Window can be constructed by a combination of inflating the View objects from an XML layout file and dynamically creating them from code. View objects defined in the XML Layout files that need to be referenced from code have an id attribute. The id attribute is a string that can be used to identify the View in the code. When the Android application is compiled it creates a file called the R file to bind each of these id attributes to a unique int value. The value of this id can be used in the script to refer to the View. Views that are dynamically created from the code on the other hand do not have an id attribute set. To reference these Views from the code, we need infer a name from the state of the View. For a Button for example we use its label text.

Physical buttons on the device are not represented by View objects in the hierarchy. To reference them from the script name have been reserved for them. Table 1 shows the reserved names for the supported buttons.

Physical button	Script Name	Script Actions
back	backButton	press()
home	homeButton	press()
menu	menuButton	press()
power	powerButton	press()
volume	volumeButton	up() down()

Table 1: Reserved names for Physical Buttons

When a UI event is triggered from the input script, it is passed to the WindowManager model. The WindowManager keeps a reference to the currently visible Window. If we can identify the View by a unique name we can traverse the View hierarchy in the current Window to locate the actual View object by its name. If the view has a registered listener corresponding to the event fired we can directly trigger the listener. If the view has no listener registered the event is passes to the Activity to be handled.

2.2 System Events

Android applications are not only driven by user events but also by events from the Android OS. These events are called system events. System events include notifications of the WiFi connection dropping or the GPS location changing. An application can register to be notified of specific system events. Depending on the event type an application can either register a Broadcast Receiver to be executed when the event occurs or register a callback on the corresponding service manager.

One way to test the behavior of an application triggered by system events is by manually performing actions to induce them on an actual device / emulator. For example switching off the device's WiFi connection can simulate the WiFi connection dropping. We can also reboot or rotate the device to induce boot completed and orientation change system events. This is the approach used by UI Automator [2]. It allow the user to script any actions the user would normally perform on the device and then executes them one by one on the emulator / device.

But, some events like battery low notifications, triggering an alarm or testing the behavior of the application using specific input event parameters is not possible using this approach. To simulate specific events such as incoming calls, location changes and battery levels being low, we can use of the Android Debug Bridge (ADB) tool shipped with the Android SDK. This tool allows us to interact with a shell on the emulator. The shell provides us with the functionality to set the network state, battery state, locating and simulate incoming calls. This method is used in Dynodroid [16] to simulate system events.

The Android JUnit testing framework does not currently support programically changing the state of the system or the sending of system events form the test cases. The battery state and network status for example has to be set on the device before the tests are run.

As we mentioned above, certain application behavior can only be triggered by system events. This is why it is so important that we find a way to simulate all kinds of system events. But we also need to consider that if the system broadcast a specific state change, the application can querying 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. This is not a problem for the tools running on the emulator as the actual network manager service will be running and will make sure that the network state is changed. What this means for JPF-Android is that it can not just allow events to be sent from the scripts directly to the application. We need to intercept some of these events and forward them to the relevant service manager to make sure the its state also reflects the change.

JPF-Android allows users to script system events by constructing Intents in the script that describe specific events. These events are then parsed by the ActivityManager natively and sent to the relevant service manager. Network status change events for example are sent to the Network Manager and battery status events are sent to the battery or the power manager. The manager parses the Intent and updates its status. It then send a broadcast to the application indicating a change in its state and back to applications registered for the event.

JPF-Android also predefines common Intents that are used frequently to simplify the scripting. One of the advantages of JPF-Android is that because we are modeling the Android OS it is possible for us to easily trigger intricate system events such as triggering a timer or alarm from the script.

2.3 Application Verification

Many of the dynamic testing tools for Android applications makes use of code coverage analysis as an indication of the effectiveness of the tool [16, 11, 7, 12]. Code coverage analysis measures which code of the application was executed during testing [10]. The higher the coverage, the larger amount of code was executed. If code was not covered by the testing tool, there is no chance of detecting any errors in it. Although code coverage is a good measure of how thorough the testing tool executed the application's code, it has some weaknesses. For example, it can not verify that the correct code was covered (or not covered) for a specific test case and in which order the code was covered. It requires the tester to manually inspect the coverage of the code after each test run to make sure all the correct code was covered.

JPF-Android introduces Checklists as a way to automatically verify a specific application flow during testing.

The user can register and unregister Checklists in the input script. A Checklist consists of an ordered list of Checkpoints in the code that has to be visited before the Checklist is unregistered. A Checklist can also contain negative Checkpoints that indicate that a certain Checkpoint is not allowed to be reached. Checkpoints is a user created method annotation in the application's source code. Each Checkpoint annotation has to have a unique name.

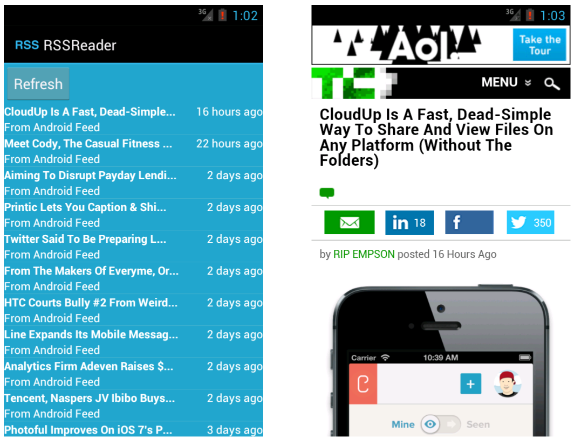
Checklists can be used to verify that certain methods are executed and in a specific order. This is useful as it provides an automatic way to verify that an application executed as expected during each test run. Checklists is not a replacement for code coverage but an enhancement. That is why JPF-Android also supports code coverage calculation using Java PathFinder's built-in coverage calculation listener.

The one limitation of our current Checklist implementation is that it only support Checkpoints on code executed by the main thread of the application.

Some of the other testing tools provide another type of verification in the form of assert statements [18, 6, 7, 11]. Assert statements allow the developer to assert the correctness of certain system properties during testing. This includes properties such as UI element's size, position and state and input / output values such as Intents coming in and going out of application components.

We are currently working on extending this type of property verification in JPF-Android to verify not only UI but also other application properties.

3. EVALUATION



CloudUp Is A Fast, Dead-Simple... From Android Feed	16 hours ago
Meet Cody, The Casual Fitness ... From Android Feed	22 hours ago
Aiming To Disrupt Payday Lendi... From Android Feed	2 days ago
Printic Lets You Caption & Shi... From Android Feed	2 days ago
Twitter Said To Be Preparing L... From Android Feed	2 days ago
From The Makers Of Everyme, Or... From Android Feed	2 days ago
HTC Courts Bully #2 From Weird... From Android Feed	2 days ago
Line Expands Its Mobile Messag... From Android Feed	2 days ago
Analytics Firm Adeven Raises $... From Android Feed	2 days ago
Tencent, Naspers JV Ibibo Buys... From Android Feed	2 days ago
Photoful Improves On iOS 7's P...	3 days ago

CloudUp Is A Fast, Dead-Simple Way To Share And View Files On Any Platform (Without The Folders)

by RIP EMPSON posted 16 Hours Ago

Figure 3: Example of using an urlInputIntent to associate an URL with a specific input file

In the evaluation section, we look at a few examples of how JPF-Android can be used to verify some properties of an RSSFeed reader application. The RSS feed reader Android application downloads and stores RSS feed items in a database. The most current items are then displayed in a combined list of RSS feed items ordered by creation time. When an item in the list is clicked a webview is shown containing the item's contents. Figure 1 shows the application displaying the current RSS feed items and then the webview displaying the link to the feed item.

We especially choose a example application utilizing a network connection, SAX XML parser, a database connection and also three of Android's main components namely an Activity, Service and Broadcast Receiver. JPF-Android is implemented to enforce the strict life cycle of each of these Android application components.

JPF-Android aims to allow the user to script as much of the external input to the application to ensure deterministic input values. To simulate a network connection, JPF-Android allows the user to associate an URL with a filename containing the network data in the input script. In figure 1 a specific predefined intent, called an urlInputIntent, is constructed and sent to the system from the script. When a connection is opened to this URL, JPF-Android reads its input from the file. This input stream is then sent to a SAX XML parser to be parsed.

All code that is executed on the JPF JVM, is model-checked. To avoid unnecessary model-checking of the entire SAX parser, we implemented the SAX Parser as a thin client with native method callbacks run on the Oracle JVM. The native parser parses the input using a Document Object Model (DOM) Parser. The SAX parser then traverses the DOM of the document and calls the callback methods on the Default Handler.

Next, the Feed items are put into a database. The JPF-Android database model is very basic. It parses the SQL from the application using JSQL Parser and stores a list of column names and list of values for each table. The database model will later be enhanced to also allow the user to use the script to specify files containing a specific database state.

We will now look at two scenarios in which JPF-Android can be used to verify the flow of the RSS feed reader application.

3.1 Scenario 1

When the user clicks on the refresh button in the main Activity, the application must retrieve the newest items from the 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, the user must be notified that the WiFi is disconnected and the application could not update the feed. To verify that our application correctly handles this situation, we use the script extract in figure 2.

This scrip extract in figure 2 registers a Checklist to verify the sequence of events after the WiFi drops. In line 2 an WifiOffIntent is broadcast to the system. This Intent is intercepted by the ConnectionManager and it responds by setting its WiFi Connection to the disconnected state. If the application now requests the status of the WiFi connection from the ConnectionManager, it will report that the WiFi is disconnected. Line 3 in the script then simulates an click of the reload button. The reload button click fires the onClickListeber on the button. This listener checks if the WiFi is connected. If it is disconnected, it notifies the user and does not try to run an update. Line 4 unregisters the Checklist for further notifications.

As mestioned before JPF-Android waits until the Android application has no more application events to handle before it retrieves a new events from the input script. When the reload button event is fired the main thread of the Application will firstly respond to this button click and only when the application is idle again, will line 4 be executed to unregister the Checklist. In other words we can safely assume that if the update has not been run before the deregistration on the checklist, it will not run.

In the case that the application incorrectly tries to update even though the WiFi is disconnected, the second checkpoint "!runUpdate" will fail as the application will

```
 1.  @urlInputIntent.putExtraString("url","http://feeds.feedburner.com/Mobilecrunch.rss")   2.  @urlInputIn-
tent.putExtraString("file","src/Mobilecrunch.rss")
 3. sendBroadcast(@urlInputStreamIntent)
```

Figure 1: Example of sending a "urlInputIntent" to associate an URL with a specific input file

```
1. registerChecklist("Wifi Down NO Update", "clickReload","!runUpdate","notifyUserWifiOff" )
2. sendBroadcast(@WifiOffIntent)
3. $buttonReload.onClick()   4. unregister("Wifi Down NO Update")
```

Figure 2: Example of using Checklist to verify that an update it not run while the WiFi is off

try to update. In this case JPF-Android will report the violating checklist at the end of the application execution (See figure 3.1)

3.2 Scenario 2

When the WiFi comes up after it was disconnected, we want to ensure that the RSSFeed application updates its feed. The application registers a Broadcast receiver that listens for when the WiFi is connected again and then launches an Update.

Figure 4 shows the extract from the input script for this scenario. In line 1 a Checklist "Wifi-Up - Update" is registered that verifies that after the WiFi reconnects, an update is initiated.

When the "WifiOnIntent" is broadcast, it is intercepted by the ConnectionManager which sets the state of the WiFi connection to Connected. The ConnectionManager then re-broadcast the Intent to the application which is registered for network status changes. The application's Broadcast receiver is then notified about the change and it responds by updating its feed.

If their was an error in the application logic, for example, and the update is never run, JPF-Android will report the checklist at the end of execution (See figure 3.2).

4. RELATED WORK

We have mentioned that the two challenges of testing Android applications are the finding relevant input sequences and finding a way to execute the application. This section looks at how other projects approached these challenges.

Android's monkey tool is a stress testing tool that takes a black-box approach to Android application testing [12]. It automatically generates pseudo-random input events and fires the events while the application is running on the emulator / device. Monkey can simulate UI events and a few system device events such as changing the orientation of the device or pressing the volume button. Input events can be generated quite quickly as they are randomly selected, but it has the disadvantage of being slow as it runs on the emulator. Generating events randomly is a useful way of stress testing as it has a good chance of including at least a few sequences beyond normal functionality of application. But, as monkey has no knowledge of application, it can also generate many unimportant event sequences and repeat unimportant events while not triggering important events. A study by Machiry, Tahiliani and Naik [16], showed that Monkey focuses mainly on UI event and that many as 30000 input events from monkey only covered 60% of the application's code. Hu and Neamtiu [14] combined Monkey with code instrumentation to generate logs that can be analyzed for errors.

Android also released a JUnit testing framework for the Android emulator [?]. This framework allows the developer to write unit and integration tests for Android applications. It supports Android component testing by providing the application with a mock context. It also supports testing the interaction between Activities and Services be providing annotations such as @flaky to indicate that there might be timing issues and the test case must be run a few times before failing. 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. The Android JUnit tests is run on the emulator which means that all of the tests have to be converted to Dalvik bytecode. As a result these tests run very slow and are not suitable or practical for test driven development. 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.

MonkeyRunner provides a python API for users to script input event sequences that are simulated by the Android emulator. The MonkeyRunner script includes functionality to install a package, start Activities , trigger specific key and touch events and taking screen shots. These screen shots can be used to view and compare results. It also allows the script to request system properties such as the height and width of the device and adapt its script around it. As it runs on the emulator it can actually return the correct values for these properties.

Dynodroid makes use of the same approach as MonkeyRunner where it sends input events to the OS using the ADB daemon in the device to fire UI and system events [16]. Its main approach it not to let the user script these input events, but, automatically choosing them by using an observe, select and execute approach. This approach improves on monkeys random selection approach. 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. Although Dynodroid's approach for selecting the input events is very successful, it is dependent on a custom version of the emulator for running the test input. It also allows the user to stop the execution and input their own events, but these sequences are not saved for future runs.

Robolectric is a JUnit testing framework for Android that 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 [18]. Robolectric

```
==================================================== violating CheckLists
Checklist: ''Wifi Down NO Update''
Line: 1
List: [clickOnReload, -->!runUpdate, notifyUserWifiOff]
Reason: Failed because invalid checkpoint ''runUpdate'' reached.
==================================================== results
```

```
1. registerChecklist("Wifi Up - Update", "WifiUp","runUpdate")
2. sendBroadcast(@WifiOnIntent)
3. unregister("Wifi Up - Update")
```

Figure 4: Example of using Checklist to verify that an update is run when the WiFi reconnects.

was specifically developed to support test driven development as allows android test cases can execute within seconds on the JVM. Like JPF-Android, Robolectric also suffers from the problem that if Android changes, it must be adapted. It focuses on testing the application using JUnit test cases and including custom assert statement for Android functionality.

Model-based testing approaches extract a model of the behavior of a system and then utilizes this model to generate correct and meaningful test cases or input values to test the system [9]. Various projects [20, 8] have developed solutions to extract a model from an Android application and use this model to either generate test cases that can run on the emulator.

Other projects focus on applying symbolic verification techniques such as concolic testing [15] or symbolic execution [17] to automatically generate valid inputs while traversing the application.

Lastly Android Mock [4] and [5] are mocking frameworks specifically adapted for Android application JUnit testing on the emulator. Although mocking can be in cases that we use external libraries, it provides us with limited support in terms of the actual functionality of the model. It only support stubbing the class and return specific values.

5. LIMITATIONS AND FUTURE WORK

The Android software stack is a very large and complicated system. Before modeling the system we have to understand exactly how the original system works to be able to create a reliable model. Although our models are still limited, they are modeled accurately. We are currently still working on expanding out model base.

A limitation of JPF-Android is that it is designed to only support the testing of a single application. This allows us to simplify the models of the Android software stack greatly. Although we only support a single application the components of an application can still be tested by calling them from the test script.

Android applications have many external sources of input such as the database, the data sent over the network and the location data. These external input sources can provide non-deterministic input values. We want to the input sequences to always result in the save output for each run so that the script can be used for integration testing. Currently we are working on including elements in our script to support the injection of these external values into the system. This will allow the user, for example, to inject data into the database from inside the input script.

The largest limitation of JPF-Android is its 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.

6. CONCLUSIONS

In the paper we presented an approach to verify Android application directly on the JVM outside of the emulator. Most of the other testing frameworks we discussed run their tests on the emulator/device with limit resources. Testing the application on the JVM provides the tool with a large amount of system resources and allows it to save testing time. Especially when using a test driven development approach where tests are created first and run many times during the development of a project, testing time has to be minimal.

Our approach allows the user to script compact event sequences containing GUI events as well as system events. Most the currently available systems focus only on GUI testing and due to this, can not test all of the application's functionality.

Lastly we provide the user with a new way to verify the flow of the application. As seen in the evaluation section, this allows the user to verify that a system adheres to its behavioral specifications of the application.

7. REFERENCES

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

[2] Ui automator. `http://developer.android.com/tools/help/uiautomator/index.html`. Accessed: 19 July 2013.

[3] Android lint, November 2007. `http://tools.android.com/tips/lint/`. Accessed: 17 July 2012.

[4] Android mock, November 2007. `http://code.google.com/p/android-mock/`. Accessed: 17 July 2012.

[5] Mockito, November 2007. `http://code.google.com/p/mockito/`. Accessed: 17 July 2012.

[6] User scenario testing for android, November 2007. `code.google.com/p/robotium/`. Accessed: 17 July 2012.

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

[8] D. Amalfitano, A. R. Fasolino, and P. Tramontana. A gui crawling-based technique for android mobile

```
======================================================= violating CheckLists
Checklist: ‘‘Wifi Up Update’’
Line: 1
List: [networkStatusChange, -->runUpdate]
Reason: Checkpoint ‘‘runUpdate’’ not reached before deregistration.
======================================================= results
```

application testing. In *Software Testing, Verification and Validation Workshops (ICSTW)*, pages 252 – 261, March 2011.

[9] L. Apfelbaum and J. Doyle. Model based testing. In *Software Quality Week Conference*, pages 296–300, 1997.

[10] S. Cornett. Code coverage analysis. *Bullseye Testing Technology*, 2002.

[11] A. Developers. Monkeyrunner, June 2011. `http://developer.android.com/tools/help/monkeyrunner_concepts.html`. Accessed: 04 July 2013.

[12] A. Developers. Ui/application exerciser monkey, June 2011. `http://developer.android.com/tools/help/monkey.html`. Accessed: 04 July 2013.

[13] D. Hackborn. Multitasking the android way. [article], April 2010.

[14] C. Hu and I. Neamtiu. Automating gui testing for android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST ’11, pages 77–83, New York, NY, USA, 2011. ACM.

[15] 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)*, July 2013.

[16] R. N. M. Machiry, Aravind Tahiliani. Dynodroid: An input generation system for android apps. *SIGSOFT Softw. Eng. Notes*, Aug. 2013.

[17] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood. Testing android apps through symbolic execution. *SIGSOFT Softw. Eng. Notes*, 37(6):1–5, Nov. 2012.

[18] B. Sadeh. A study on the evaluation of unit testing for android systems. *International Journal of New Computer Architectures and their Applications (IJNCAA)*, 4(1), Sept. 2012.

[19] H. van der Merwe, B. van der Merwe, and W. Visser. Verifying android applications using java pathfinder. *SIGSOFT Softw. Eng. Notes*, 37(6):1–5, Nov. 2012.

[20] W. Yang, M. Prasad, and T. Xie. A grey-box approach for automated gui-model generation of mobile applications. In V. Cortellessa and D. VarrÃş, editors, *Fundamental Approaches to Software Engineering*, volume 7793 of *Lecture Notes in Computer Science*, pages 250–265. Springer Berlin Heidelberg, 2013.