Environment Modeling Using Runtime Values for JPF-Android

Heila van der Merwe
CAIR, Meraka, CSIR
Dept. of Computer Science
University of Stellenbosch,
South Africa
hvdmerwe@cs.sun.ac.za

Oksana Tkachuk
SGT Inc. / NASA Ames
Research Center
Moffett Field, California
oksana.tkachuk@nasa.gov

Sean Nel, Brink van der Merwe and Willem Visser
Dept. of Computer Science
University of Stellenbosch,
South Africa
{17903262, abvdm, wvisser}
@sun.ac.za

ABSTRACT

Software applications are developed to be executed in a specific environment. This environment includes external/ native libraries to add functionality to the application and drivers to fire the application execution. For testing and verification, the environment of an application is simplified/abstracted using models or stubs. Empty stubs, returning default values, are simple to generate automatically, but they do not perform well when the application expects specific return values. Symbolic execution is used to find input parameters for drivers and return values for library stubs, but it struggles to detect the values of complex objects. In this work-in-progress paper, we explore an approach to generate drivers and stubs based on values collected during runtime instead of using default values. Entry-points and methods that need to be modeled are instrumented to log their parameters and return values. The instrumented applications are then executed using a driver and instrumented libraries. The values collected during runtime are used to generate driver and stub values on-the-fly that improve coverage during verification by enabling the execution of code that previously crashed or was missed. We are implementing this approach to improve the environment model of JPF-Android, our model checking and analysis tool for Android applications.

Categories and Subject Descriptors

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

Keywords

Android Application, Environment Generation, Verification

1. INTRODUCTION

Software applications are difficult to analyze and verify because their execution directly and indirectly depends on their environment. This environment includes the hardware configuration, the I/O devices available, the architecture of the CPU, as well as the operating system, drivers, libraries and even other external applications. Applications can react differently for different configurations. Therefore, to ensure the stability of an application, it must be thoroughly analyzed and tested resulting in a large number of tests.

In order to dynamically test, verify or analyze the behavior of an application, we limit the behavior and state of the environment using mocks, stubs and models. One approach to environment modeling is to manually create generic models that can be reused by many applications and improved and extended over time. Developing generic models is difficult since it requires in-depth knowledge of the internals of the component.

NetIO [2] and NetStub [1] are tools that model the network library for Java applications. NetStub provides manually created stubs for distributed Java applications and supports capturing the interaction between the applications for use as a driver during unit verification. NetIO allows a single Java application (either client or server) to run on Java PathFinder (JPF) [9] and interact with its counterparts running external to the verification tool. This interaction is cached to allow JPF to backtrack of the state-space.

The modeling process can be assisted by using automatic model generation tools such as OCSEGen [6]. OCSEGen uses static analysis of Java byte-code to create stubs retaining certain side-effects and return values. It can generate models returning three types of values: default, choice or symbolic values. Returning default values can miss interesting behavior of the application, but it works well for cases where the application component calling the stub does not rely on the content of the return value. Alternatively, generated methods can return a set of all possible return values, but this results in too many possibilities to verify. Symbolic objects can also be returned by the stubs if the application is run on a symbolic execution tool such as Symbolic PathFinder (SPF) [4]. However, OCSEGen has limitations typical to static analysis: it may produce over-approximation of side-effects and cannot analyze native code.

Another tool, nHandler [5], automatically models methods of libraries / applications on Java PathFinder (JPF) [9]. It delegates the method execution to an instance of the object in its actual environment on the Java Virtual Machine (JVM) and then converts and returns the result from the modeled method in the JPF environment. This approach assumes that the object can be instantiated and run directly on the JVM, which is not the case for Android applications since their external libraries are not available outside of the Android environment.

This approach, however, explores how values observed during dynamic analysis can assist driver / stub generation for testing and verification purposes. We implement our approach for Android applications with many external dependencies that need to be modeled, to verify the application in a closed, finite state environment. We show how our approach can enable the execution of application code that cannot otherwise be executed using the default stubs and driver of our Android application analysis tool, JPF-Android [8].

2. BACKGROUND

2.1 The Android platform

Android applications consist mainly of Java code. They are compiled against a stubbed version of the Android library before being converted into DEX byte-code and uploaded to a Android device.

Android applications consist of a set of loosely coupled components, built on top of an extensive application framework. The framework provides the core implementation of an application. Android applications consist of four main components[1]. *Activities* are used to construct and control the different windows of the application. *Services* are used to run operations not associated with a user interface (UI), such as data processing and network connections. *BroadcastReceivers* (BR) implement a publisher-subscriber design pattern to allow applications to subscribe to system events in the form of *Intents*. *ContentProviders* perform create, read, update and delete (CRUD) operations on content stored in files or databases and share this content between applications. These components are all executed by the main thread of the application in response to messages received from the Android system server.

Android applications are executed in their own Linux process, sand-boxing their execution on the Dalvik VM and their application data from other applications and from the system. The Java native interface (JNI) allows Android applications to call native C/C++ methods from the Java code. Local and remote services often call native code to communicate with drivers written in C/C++.

The Android system server consists of multiple concurrently running services. These services process user and system events generated by native drivers (display- and network drivers) as well as Android applications. These services drive applications by placing messages on the applications' message queues to be processed by their main thread. All data/messages sent between the application and system services need to cross process boundaries. This communication is managed by the native Binder driver which stores unique references for all Binder services.

2.2 Modeling the Android environment

In our previous work we developed JPF-Android — a model checker and analysis tool for Android applications [8] built on JPF. The purpose of JPF-Android is to enable Android applications to run in an environment that allows us to track their execution and listen for specific property violations. In this approach, more accurate results are obtained by executing a greater proportion of application code.

Android applications are designed to run on the Android software stack, which is only available on Android devices. To run the applications on JPF, an extensive environment model is required to create a closed, finite state system.

At a high level, the environment of an application consists of two main components: the driver (which calls the application code) and the stubs (which model components called from the application) [6].

JPF-Android provides a default driver that generates and fires all possible input event combinations by making use of static and dynamic analysis to detect entry-points of the application at runtime. The driver currently makes use of default arguments for these generated events since determining more suitable values requires a more complex analysis such as symbolic execution.

JPF-Android includes many environment models created manually for components required by the application such as file management, XML- and resource parsing, network connections and database interactions. It also includes automatically generated default stubs generated using OCSEGen for components we are not interested in verifying at this time or could not analyze due to native method [7]. These stubs include components related to the GUI drawing implementation, OpenGL, animation and many more. There many components that need to be modeled and we are creating these stubs on a per app basis and adding them to the tool as required.

[1] `http://developer.android.com`

2.3 Motivating example

Listing 1 shows the `MusicReceiver` BroadcastReceiver component of the RandomMusicPlayer Android application implemented to respond to media button presses on the physical device.

```
1 public class MusicReceiver extends BroadcastReceiver {
2
3  @Override
4  public void onReceive(Context ctx, Intent intent) {
5
6   String action = intent.getAction();
7   if (action.equals(Intent.ACTION_MEDIA_BUTTON)){
8
9     Bundle bundle = intent.getExtras();
10    KeyEvent keyEvent =
       (KeyEvent)bundle.get(Intent.EXTRA_KEY_EVENT);
11
12    switch (keyEvent.getKeyCode()) {
13     case KeyEvent.KEYCODE_MEDIA_PLAY_PAUSE:
14       ctx.startService(new Intent(...));
15       break;
16     case KeyEvent.KEYCODE_MEDIA_PLAY:
17       ctx.startService(new Intent(...));
18       break;
19     case KeyEvent.KEYCODE_MEDIA_PAUSE:
20       ctx.startService(new Intent(...));
21       break;
22    }
23   }
24  }
25 }
```

Listing 1: Extract from RandomMusicPlayer

```
1 <receiver android:name=".MusicReceiver" >
2  <intent-filter>
3   <action android:name="android.intent.action.MEDIA_BUTTON
      " />
4  </intent-filter>
5 </receiver>
```

Listing 2: Extract from AndroidManifest.xml

BroadcastReceivers are registered for specific Intent objects by using *IntentFilters*. These filters can be extracted from the AndroidManifest.xml file (Listing 2) when the application is installed or they can be specified when registering a BroadcastReceiver dynamically in the code. IntentFilters specify the action, categories and data that an Intent has to match to be forwarded to the BroadcastReceiver. Intents can also contain any number of extra Java objects stored in the *Bundle* object of the Intent. The Bundle stores a map of String-Object pairs containing extra information about the event (see Listing 3).

The MusicReceiver's `onReceive()` method is an entry-point of the application and is called by the JPF-Android driver (Listing 1). Using static analysis we parse the AndroidManifest XML file to determine the action and categories of Intents that must be fired. In the above example the action of the Intent must be set to MEDIA_BUTTON.

```
1 public class Intent implements Parcelable, ... {
2     private String mAction;
3     private ArraySet<String> mCategories;
4     private Bundle mExtras; // map of extra objects
5     ...
6 }
```

Listing 3: Extract from Intent.java

JPF-Android has no way of determining what objects should be contained in the Bundle of the Intent and so keyEvent is set to "null" on line 10 of Listing 1. The code then crashes on line 12 due to a null-pointer de-referencing exception. In addition, symbolic execution tools such as SPF have difficulty analyzing this code because of the complex structure of the Intent and Bundle objects.

```
1   if (android.os.Build.VERSION.SDK_INT >= 8) {
2     mAudioFocusHelper = new AudioFocusHelper(
       getApplicationContext(), this);
3   } else {
4     mAudioFocus = AudioFocus.Focused;
5   }
```

Listing 4: Extract from RandomMusicPlayer

In Listing 4, our next example, VERSION.SDK_INT is a static field of the `android.os.Build` class set by calling a native method in the `android.os.SystemProperties` class. The model of the SystemProperties class needs to return an Android SDK version of 8 as well as another version to ensure both branches of this if-statement are executed. Our current model of SystemProperties is a default stub and returns an empty String for String properties requested. Instead of returning default values, we want to automatically look up possible return values for native methods and then execute the code in Listing 4 non-deterministically for build version "8" as well as another valid Android SDK version. Symbolic execution can detect these values, but it requires environment generation to analyse the method as well as stub generation to create a model using the values it detects.

3. OVERVIEW OF OUR APPROACH

This research explores generation of drivers and stubs using dynamically collected method parameters and return values for testing/analysis. To collect these values we instrument methods in the application (and its libraries) to log their input parameters and return values when run in their original environment. More specifically, the following information is recorded for each method:

Application name By storing the application name, we can collect logs over many runs of the application and combine their inputs/results.

Unique run number The run number allows us to filter the logs printed during a specific run of the application.

The class signature The class information allows us to distinguish between methods with the same name.

The method signature This is used to identify the method that printed the log entry.

Input parameters Input parameters are used as "extra information" to filter the results of a method. The input parameters are also used to generate more accurate input values for entry points.

Return value We want to retrieve these values to use in method stubs.

Java code is injected at the start of a method as well as before each of its return statements to record these values. The code injected at the start of a method copies/caches the state of the parameters. It is necessary to record them before they are updated in the method. Before each return statement, a statement is injected to log the information collected about the method.

The parameters and return objects of a Java method need to be serialized to binary, XML or JSON representations. This enables us to record the state of a object and de-serialize the state back into a new object at a later stage.

The instrumented application is run in its original environment using a driver or test cases. The logs from multiple runs are collected, parsed and stored in a database where they can be searched and queried. Since the information we are logging is structured, we can use a regular expression matcher to parse the logs into objects that can be stored in an object store or database.

These logs can be queried for the parameters and return values of a specific method. If more than one return value is stored for the same input parameters, we can return them non-deterministically by the stub. Otherwise, if no values have been stored we return a default value. We use these values to improve the driver and stubs to produce a higher coverage of the application during a follow up analysis with JPF-Android.

4. IMPLEMENTATION

To apply our approach to Android applications we need to instrument the entry-points of Android applications to collect input parameters to use for event generation, as well as instrument methods that need to be modeled to collect parameters and return values for creating stubs. The main purpose of implementing this approach is to improve the coverage of the application code by enabling the execution of code that could previously not be executed with JPF-Android.

4.1 Logging using Flowlogger

To simplify logging from the application, we created an application called Flowlogger. Flowlogger contains static methods that are called from the injected code to serialize the parameters and return values and to log method information using the Android logging framework. The application name, class signature and method signature are passed to Flowlogger from the application as String values. We use XStream[2] to serialize the input parameters and return values. XStream allows serialization/de-serialization of Java objects into XML, JSON or binary data. It can both access and set all public, protected and private fields of a class or inner class. Additionally, the user can create custom converters for objects that contain context specific fields that should not be stored or used for matching. Flowlogger and XStream are injected into the libs folder of the application and transformed into DEX byte-code during the build process.

4.2 Instrumentation of the byte-code

The most popular approach to instrument Android applications is to use tools such as SOOT[3] and Androguard[4] to instrument

[2] `http://x-stream.github.io`
[3] `http://www.sable.mcgill.ca/soot`
[4] `https://github.com/androguard/androguard`

the DEX byte-code. But since DEX byte-code differs from Java byte-code, we decided to use BCEL[5] to instrument the Java byte-code generated at an intermediate phase of the application build process.

The application methods are instrumented to perform three tasks. Firstly, invoke a static method call on Flowlogger at the start of the method to serialize its parameters and store it in a String variable. Secondly, inject byte-code at the position of each return statement to serialize the return object. Lastly, send the method information to Flowlogger to log and return.

Instrumenting native methods is not so simple, since they have no Java implementation. Their method signatures must also be kept intact since it is used to automatically map native method definitions to their C++ implementation. To instrument these methods, we inject a *shadow method* for each native method in Java. A shadow method calls its native method but also logs the input and return values of the native method. All calls to the original native method are then updated to call the shadow method.

4.3 Collecting and parsing the logs

The instrumented application is run on the emulator using monkey[6], an automatic event generation tool for Android. The log statements generated by Flowlogger are retrieved by connecting to the Logcat service running on the device.

Logs over different runs and for different applications are parsed and stored in a database. We use Logstash[7] to parse and filter the logs with specified patterns into JSON objects and output them to an Elasticsearch instance. Elasticsearch is highly scalable and a fast JSON object store with an extensive Restful API built on Apache Lucene. It is commonly used together with Logstash to perform real-time analysis to monitor the state of large live systems consisting of many servers. Elasticsearch enables us to store and index our logs and obtain results for queries against these logs. To simplify these queries, we developed a Java wrapper for the Restful API that allows us to query the store for the values we require.

4.4 Using the log information

Once logs have been processed and stored, we can generate stubs that look up one or more return values by querying the Elasticsearch instance. If there are no recorded results for a method, it returns a default value.

To de-serialize and instantiate objects from XML we use XStream. Some classes in the Android application framework cannot be instantiated on the JVM directly since their fields are initialized using native methods. However, we require this initialization when translating them to JPF Objects. In the JPF environment native methods can be intercepted and modeled using the Model Java Environment (MJIEnv). On the JVM however, calls to native methods crash since their native code is not available outside of the Android environment. To avoid this from happening, we replace all native methods with default stubs instead. This allows construction of the required object instances on the JVM.

These instances are subsequently transformed to JPF objects kept on JPF's internal JVM. JVM objects can be transformed using the JVM to JPF object converter part of nHandler to create a new instance of the object on the heap in the JPF environment. The same converters can be used to transform the parameters of a method from JPF to JVM objects to look up the return value of a method.

[5] https://commons.apache.org/proper/commons-bcel
[6] http://developer.android.com/tools/help/monkey.html
[7] https://www.elastic.co/products/logstash

5. RESULTS

We are applying this approach to improve our environment model of Android applications running on JPF-Android. One of the examples we use is the RandomMusicPlayer Android application shipped with the Android SDK discussed in Section II. The RandomMusicPlayer application obtains high coverage results using dynamic testing tools such as monkey and Dynodroid [3], which ensures that we can collect a good set of input parameters and result values.

```
1 public class MusicReceiver extends BroadcastReceiver{
2
3   public void onReceive(Context ctx, Intent intent) {
4
5    Object[] array = { ctx, intent };
6    String paramStr = FlowLogger.getParamString(
7     "(Landroid/content/Context;Landroid/content/Intent;)V"
       , array);
8
9    // original method body
10
11   FlowLogger.logMethod("com.example.android.musicplayer.
       MusicReceiver", "onReceive",
12    "(Landroid/content/Context;Landroid/content/Intent;)V"
       , paramStr, null);
13  }
14 }
```

Listing 5: Instrumented MusicReceiver

Listing 1 shows the MusicReceiver of RandomMusicPlayer containing the `onReceive()` method. When the application is compiled, the MusicReceiver is instrumented to log its input parameters (see Listing 5). We created a custom converter that returns no XML for Context objects, since the Context is highly dependent on the application and its environment.

```
1 <android.content.Intent>
2   <mAction>android.intent.action.MEDIA_BUTTON</mAction>
3    <mExtras>
4      <mMap>
5        <mArray>
6         <string>android.intent.extra.KEY_EVENT</string>
7         <android.view.KeyEvent>
8            <mDeviceId>-1</mDeviceId>
9            <mKeyCode>126</mKeyCode>
10           ...
11        </android.view.KeyEvent>
12       </mArray>
13       <mSize>1</mSize>
14     </mMap>
15   </mExtras>
16   ...
17 </android.content.Intent>
```

Listing 6: XML generated by XStream for an Intent object

The application was executed on the emulator using monkey for 5000 events and we collected 11 log entries generated by the method. An extract from a log entry containing the input parameters is given in Listing 6. In Listing 6, the map of extras in the Intent

object contains an `android.view.KeyEvent` object with key code 126 which maps to a KEYCODE_MEDIA_PLAY event.

We wrote a Java application that can look up the set of input parameters given a method name. The parameters can automatically be converted into Java objects using XStream.

The next step is to extend the JPF-Android driver to look up input parameters on the fly for input events. If the driver makes use of the Intent object generated from the XML in Listing 6 as the parameter, the MusicReceiver can fire without crashing on line 12 and cover line 16 in Listing 1. The more dynamically fired events can be collected, the better coverage can be obtained for this example.

The next example is the `get_native()` method in the `android.os.SystemProperties` class contained in the Android library. This class natively looks up system properties from the device. It is used by the `android.os.Build` class in Listing 4 to look up the SDK version. This method is a good example for this approach since the method maps String parameters to primitive return values. Our current model for this class in JPF-Android was generated automatically and returns an empty String for all String system properties. We instrumented this method to call Flowlogger and ran the application on the instrumented platform to collect the logs produced by the method. We then used the values collected by the logs to manually improve our SystemProperties model and improve the coverage of the application by enabling both branches in Listing 4. The next step is to generate models that look up a set of return values for the `get_native()` method on-the-fly while running on JPF-Android.

6. CONCLUSION

In this research we explore how values collected during runtime can assist environment modeling. The effectiveness of the approach relies on dynamic analysis obtaining a good code coverage of the application. However, if dynamic analysis obtains good coverage results for the application, why would one want to verify the application? The advantage of verifying Android applications on a tool like JPF is that we have fine-grained analysis capabilities that enable us to run more complex analyses on the applications. This includes listening for properties such as infinite loops, memory leaks, deadlocks and performing data-flow analysis. It also allows us to prune the search space using the state matching and backtracking capabilities of the tool as well as enable us to use other tools built on top of JPF such as SPF.

Our approach only takes into account the input parameters of a method to determine and filter its return values. If the return value or input parameters of the method are dependent on the global state of the application or environment, we may return incorrect values. The approach can be extended to record more of the environment state in a log entry to allow us to return more accurate values. Furthermore, we assume that the methods modeled using this approach do not have side-effects on the application or that the side-effects have been retained by OCSEGen.

The next step in this work is to extend JPF-Android to use these values for input generation of user and system events as well as using OCSEGen to generate static stubs using the collected values. We plan to use the approach for modeling services used to look up the state of the system such as the BatteryManager, WifiManager, PackageManager and content exposed through ContentProviders. We can also extend our approach to collect values using different techniques such as manual inspection or symbolic execution and then generate models that look up these values.

Generating an environment model is a complex process and no single technique works for all cases. The key is to identify techniques that can be applied to certain components based on the requirements of the application. Default stubs are simple to generate, but do not always provide good coverage results. Symbolic execution can return all inputs/return values, but cannot always be run due to the complex nature of the objects involved. Our approach presents a fully automatic technique that forms the middle ground between these approaches. It works well in cases where the execution is dependent on the content returned from models but the implementation is too complex to currently analyze using symbolic execution.

7. REFERENCES

[1] E. Barlas and T. Bultan. Netstub: A framework for verification of distributed java applications. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 24–33, New York, NY, USA, 2007. ACM.

[2] W. Leungwattanakit, C. Artho, M. Hagiya, Y. Tanabe, M. Yamamoto, and K. Takahashi. Modular software model checking for distributed systems. *Software Engineering, IEEE Transactions on*, 40(5):483–501, May 2014.

[3] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An Input Generation System for Android Apps. *SIGSOFT Softw. Eng. Notes*, Aug. 2013.

[4] C. Pasareanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta. Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis. *Automated Software Engineering*, 20(3):391–425, 2013.

[5] N. Shafiei and P. Mehlitz. Extending JPF to Verify Distributed Systems. In *ACM SIGSOFT Softw. Eng. Notes*, 2013.

[6] O. Tkachuk. OCSEGen: Open components and systems environment generator. In *Proceedings of the 2nd International Workshop on State Of the Art in Java Program analysis (SOAP)*, number 1, pages 2–5, 2013.

[7] H. van der Merwe, O. Tkachuk, B. van der Merwe, and W. Visser. Generation of library models for verification of android applications. *SIGSOFT Softw. Eng. Notes*, 40(1):1–5, 2015.

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

[9] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs. In *Automated Software Engineering*, volume 10, pages 203 – 232. IEEE, IEEE Comput. Soc, 2003.