Generation of Library Models for Verification of Android Applications

Heila van der Merwe
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

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

ABSTRACT

Android applications are difficult to verify and test since they have many external dependencies. To overcome this problem, environment generation can be used to create a model of the environment to simulate the behavior of these external dependencies. Creating this environment model manually is a tedious process and although there are many techniques available to generate models, the key lies in identifying how these techniques can be applied to a specific domain. In this paper we discuss two static analysis tools OCSEGen [3] and Modgen [1] and how they can be applied to the Android domain to generate models for specific parts of the environment.

Categories and Subject Descriptors

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

Keywords

Android Application, Environment Generation

1. INTRODUCTION

Open software applications or components with external references and dependencies are notoriously difficult to test and verify. For the system-under-test (SUT) to execute soundly, a stable, controlled environment is required that reacts correctly and consistently. This is not always possible since we may not be able to control the state or behavior of the external dependencies during analysis. This problem can be addressed by modeling the environment to simulate the behavior of the external dependencies.

In testing, mocks or stubs are used to model the functionality of external libraries or services. These models can either be created by hand, or generated automatically using static or runtime analysis. Static analysis usually produces over-approximation and runtime analysis under-approximates the behavior of the environment. However, for verification, we require precise models. This is why environment modeling in the context of verification is usually done manually.

Java PathFinder (JPF) [6] is a model checker and analysis engine that allows developers to create an environment model using modeled classes to simulate the behavior of the actual classes. JPF, together with its many extensions[1] including jpf-concurrent, jpf-awt, net-io-cache, provide models for many of the Java library classes. These models are usually written by hand, since analysis of the environment is non-trivial.
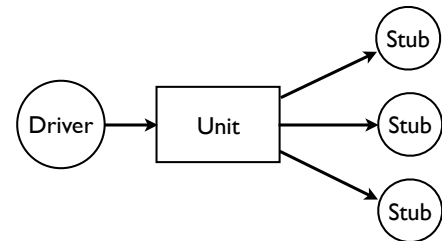
---

[1]http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects

Figure 1: Structure of an environment model

Creating these models manually is a time-consuming task that requires expert domain knowledge. The environment needs to be analyzed, simplified and optimized. Environment generation is aimed towards the automatic generation of this environment model. The system being analyzed can be viewed as consisting of two components: the *unit* under test and the *environment* including all libraries with which the unit interacts [2]. The environment can be broken down into the *driver* and the *environment stubs* as shown in Figure 1 [2]. The driver executes the unit by making calls into the application code whereas the environment stubs model the behavior of classes referenced and called from the unit [2]. Note that Figure 1 shows a simplified view of the driver and stubs. In reality the stubs might contain callbacks to the application or to the driver.

This research is work-in-progress and is based on the work by Dwyer et al. [2]. In their work OCSEGen [3], an environment generation tool, was used to generate an environment model for Swing/AWT applications. Our work is focused on generating an environment model for Android applications. An environment model is required to verify Android applications since they are heavily dependent on the Android software stack. Android applications are built on top of the Android application framework which provides the standard structure and implementation of an application. They are also event-driven and the events that drive the execution of the application are dispatched from the main Android system using Inter Process Communication (IPC).

In our previous work [4, 5] we created this environment model manually to verify Android applications using JPF. This was done by inspecting the application and then identifying the behavior required from the environment to run this application. We then mapped this behavior to a set of classes and methods that implement this functionality. The Android code-base is large and we made use of tools such as "grep" and the type & call hierarchy views in Eclipse to navigate the code. We found that the relevant classes have references and dependencies on other parts of the system or native libraries without which they would not be setup

correctly or they might not run at all.

For this research we improve our manual approach by identifying areas where the tools can replace or assist our manual effort. To do this we investigate two static analysis tools geared towards environment generation: OCSEGen [3] and Modgen [1]. These tools were chosen as we have previous knowledge about their functionality and implementation. These tools are both developed for analysis of Java applications and both tools have previously been applied to environment generation of Graphical User Interface (GUI) applications [1, 2]. We apply these tools to our approach and present some practical results for running them on the Android domain to answer the following research questions:

RQ1: Are these tools applicable to the Android domain and how can they assist in automatically generating an environment model for Android applications?

RQ2: What are the limitations of these tools and can we improve or extend their functionality to be more useful to the Android domain?

2. BACKGROUND

Modgen and OCSEGen are both built on the SOOT[2] static analysis framework which analyses Java byte-code.

OCSEGen [3] is an environment generation tool that can generate both drivers and environment stubs for the unit. The driver generation is configured by user specifications given in LTL or regular expressions. For stub generation the tool makes use of static analysis and side-effect analysis. The tool has many configurations for stub generation. It can either analyze the classes in the unit or in the environment. For both of these configurations the user can specify fields, inside or outside of the unit, to track using side-effect analysis. When the tool is run, it builds a call graph of all reachable methods from the component under analysis. This graph is then searched for statements that change the values of these fields. The changes are percolated up to the first class reachable from the unit to retain side-effects to the fields being tracked. The changes are stored in method summaries of each reachable method. The summaries are then used for code generation of stubs. For this work, we make use of the stub generation capabilities of the tool. The tool is very efficient at generating empty-stubs, but its strength lies in being able to find and preserve side-effects in the generated stubs.

Modgen [1] is an environment generation tool focused on optimization of library classes by reducing their complexity. The tool has two modes: In the first mode, it generates an empty stub of a given class by returning default values from its methods. In the second mode, it makes use of program slicing to generate an abstract model of a class. It allows the user to specify fields of a class that store values important to its functionality [1]. The class is then "sliced" in the sense that methods, fields and statements with no reference to these fields are removed or stubbed out so that the class only includes statements relevant to these specific fields. After the slicing is complete, decompilation of the sliced code is required to get the Java source code.

Android applications are built on top of the Android application framework which provides the core implementation of an application. Android applications consist of a collection of components implementing one of the four base component types exposed
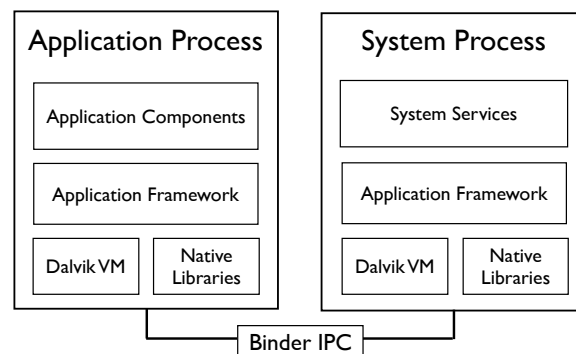
---

[2] http://www.sable.mcgill.ca/soot

Figure 2: The Android environment

by the framework: Activity, Service, BroadcastReceiver and ContentProvider. Each component type has a specific task to perform which it does by making use of local services, remote services and utility classes to store, transport and process data.

Android applications are event driven and their execution is driven by User Interface (UI) and system events. These events are dispatched from the system to the currently active application components through IPC and the Android application framework. Each Android application together with the application framework runs in its own process (Figure 2). The core functionality of the Android OS is also implemented as an Android application called the system server and runs in its own Linux process. Each process has its own memory, classloader, threads and Dalvik VM that do not influence the execution of the other processes.

3. OUR APPROACH

Android applications can also be divided into the *unit* and *environment* structure used in environment generation (See Figure 3). The unit consists of the Android application code and the environment consists of the Android application framework and the Android system. The environment can then be broken down into the *driver* and the *environment stubs*.

Android applications are driven by UI events as well system events. Application components such as the Activity and Service components follow a strict life-cycle and system events can change the state of a component by calling its life-cycle methods in a specific order. To verify the application components, they need to be executed in the way they were intended to execute on the Android software stack.

To ensure applications are exercised correctly we need to preserve important behavior of the application framework and Android system. This includes the behavior of the framework to process incoming events and manage the state of components. Instead of creating stubs for all the classes in the environment, a *core* set of classes are identified which needs to be preserved. These classes can be simplified and optimized as long as their behavior stays the same. The rest of the classes in the environment have little or no influence on the application's execution and their implementation can be stubbed out.

For this research we focus on creating general environment stubs/models that can be used to execute Android applications. To generate these stubs we follow the following methodology:
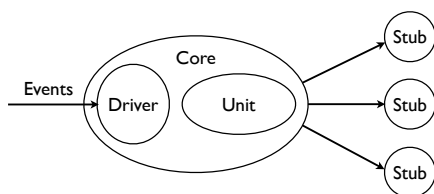
- identify the set of core classes,

Figure 3: Environment of an Android application

- find external references to the core,
- generate stubs for these external references and
- generate optimized models for the classes in the core.

3.1 Identifying the core environment

The classes in the core environment implement behavior of the framework that is crucial for correct execution of the application. The core classes can be identified by looking at the components of the Android framework and Android system on which the application depends and then selecting all classes implementing these components.

OCSEGen can be used to identify the set of classes referenced from the application [2], but there is no way to distinguish which classes referenced from the application are important enough to keep in the core.

3.2 Generating environment stubs

The generation of the environment stubs is an iterative process. We start by generating empty stubs for all classes and methods referenced from the core. This set of generated stubs must be updated if the core changes.

Although the classes outside of the core environment do not contain functionality crucial to the application execution, they might be referenced from the core or from the application and they might have side-effects on the core classes. OCSEGen allows us to specify fields in the core for which side-effects will be retained when generating the environment stubs. The Android framework is large and side-effect analysis is computationally expensive so side-effect analysis should be run on a specific feature one at a time.

3.3 Refining the implementation of the core

The set of core classes contain the main functionality required by the unit, but their implementation tends to be intricate and dependent on other parts of the environment. This is especially a problem for verification techniques such as model-checking that suffer from the state explosion problem [1]. We can optimize the execution of the core classes by reducing the complexity of their implementation and their external references to other parts of the framework. This is done by using the environment generation tools to identify and remove code that does not directly influence the execution of the application.

4. RESULTS

To illustrate our current progress, we discuss some of our findings from using the tools to create environment stubs. The application we used for the experiments is a Calculator application consisting of two Activity components. Each Activity has its own UI associated with it which displays specific operations to the user. We used this example because it is a simple GUI Android application with limited behavior, but it still requires the basic features required by all Android applications.

4.1 Identifying the core

The core environment consists of the classes modeled by hand together with classes in the application framework used without modification. We identified a set of 65 core classes from our previous work where we modeled the environment manually. A subset of the core classes, such as the base component classes, will always be included in the core, but some classes might change as the requirements of the unit changes. For a specific application we might for example require a model to make a connection over the network whereas another application might require code to interact with a database. The main components used by the Calculator application are:

Event handling As mentioned in Section 3, Android applications are driven by UI and system events dispatched by the system services. The Calculator GUI listens for click events on the `Button` widgets. To process incoming UI events, the application requires that the event listeners, properties of the widgets as well as the hierarchy of the widgets be preserved. The widget classes include the `Button`, `EditText` and `TextView` objects whereas the `Window` and `WindowManager` classes store containment of these widgets. The UI listeners can then be called directly to processes input events.

The Calculator also responds to system events that change the state of the application components. To ensure the correct execution of the application, the classes responsible for processing the system events and calling the life-cycle methods of the application components must also be preserved.

Local Services The Calculator application makes use of two local services running it its own process. The `LayoutInflator` service is used for instantiating the GUI from XML files and the `ResourceManager` is used for retrieving application resources such as images or constant values defined in XML. We modeled both of these services manually to simplify their implementation while preserving their behavior as required by the application.

Remote Services One very important remote services required by Android applications is the `ActivityManagerService`. The `ActivityManagerService` manages the state of all application component running on the device by sending system events to application. The Calculator application is started using this service and it makes use of this service to start other application components. This requires the service to respond to calls from the application such as starting a Service or Activity. This service was also modeled manually.

The main problem with automating the discovery of the core environment is that it requires domain knowledge of the system to identify which components of the framework/system need to be preserved and which classes correspond to these components.

4.2 Generating environment stubs

The set of core classes identified in the previous section was used to generate 460 empty stub classes. To ensure that we have not stubbed out classes with side-effects on the core environment, we identified the fields that are important for correct execution of each feature. We then ran side-effect analysis to see which of the environment classes modify or effect these fields.

The first example on which we ran side-effect analysis is the `SpannableStringBuilder` class. This class implements the `CharSequence` interface and acts as a wrapper class for text stored in the `mText` field of the `TextView` object (Listing 4). The `SpannableStringBuilder` stores the text as an array of primitive char objects and allows the text to be manipulated.

Although `TextView` is part of the core environment, `SpannableStringBuilder` is not. We ran side-effect analysis to identify all methods that have a side-effect on the `mText` field. Listing 1 shows a code extract from the stub of the `SpannableStringBuilder` class generated using OCSEGen.

```
1    public class SpannableStringBuilder implements CharSequence,...{
2      private char[] mText;
3      ...
4      public SpannableStringBuilder(java.lang.CharSequence param0) {
5        if (Verify.randomBool()) {
6          this.mText=((char[])Verify.randomObject("char[]"));
7        }
8      }
9      public android.text.SpannableStringBuilder insert(int param0,
          java.lang.CharSequence param1) {
10       if (Verify.randomBool()) {
11         this.mText=((char[])Verify.randomObject("char[]"));
12       }
13       return this;
14     }
15     public int length() {
16       return Abstraction.TOP_INT;
17     }
18     ...
19   }
```

Listing 1: Stub of the SpannableStringBuilder class

To generate this stub, we used *may* side-effect analysis which shows changes that *might* be made to the fields. The `Verify.randomBool()` statements indicate a non-deterministic choice and are used when verifying the application on JPF. They are used here to indicate that for some executions of the method, this side-effect might not happen. The `Verify.randomObject` statements are also used when the application is run on JPF to represent a non-deterministic choice over an object.

Side-effect analysis has difficulty to track method calls on arrays since they have no explicit Java object implementing their behavior – it is mainly implemented in native code. Arrays are usually abstracted in static analysis, i.e., the elements are not distinguished. From the resulting stub in Listing 1 we can see that OCSEGen can recognize that certain methods, such as the insert method, has a side-effect on `mText` whereas other methods such as the length method has no side-effect on this field. It can not however indicate what the effect is.

This information can be used during stub creation. It indicates that before creating an empty stub for a class, we need to inspect certain methods, such as the insert method, to ensure that we preserve possible side-effects to the core environment. This is also the case in the Calculator example where `SpannableStringBuilder` stores the text entered in the `EditText` widget. If we do not preserve this text and its changes, the Calculator using the value of this field will execute incorrectly (see lines 6, 9 and 11 of Listing 2).

Another example we looked at is the `Activity` class and its fields storing the current state of the Activity. As the state of the Calculator's Activity changes these fields need to be updated. We ran side-effect analysis on the environment to see which classes

```
1    public void onClick(View v) {
2      // retrieve the button that was pressed
3      Button button = (Button) v;
4
5      // get the digit/operator that was pressed
6      String value = button.getText().toString();
7
8      // get the current expression in the valueEdit
9      String expression = valueEdit.getText().toString();
10
11     if (value.equals("=")) {
12       // calculate result of current expression
13     }
14   }
```

Listing 2: Extract from the CalculatorActivity

outside of the core environment have side-effects on these fields. Side-effect analysis identified the `Instrumentation` class as having a side-effect on the `mResumed` field (See Listing 3).

```
1    class Instrumentation {
2      ...
3      public void callActivityOnResume(android.app.Activity param0){
4        if(Verify.randomBool()){
5          param0.mResumed=true;
6        }
7      }
8      ...
9    }
```

Listing 3: Stub of the Instrumentation class

From this result we can see that the `callActivityOnResume`-method may change the value of the Activity's `mResumed` field to true. Since we want to keep the values of these Activity fields set to the correct values, we need to preserve the behavior of this method to ensure the field is set when this method is called instead of stubbing it out and loosing this behavior. If we did not recognize this side-effect to the `mResumed` field the value of the field would have been incorrect and the Activity would execute incorrectly, since it does not reach the resumed state.

4.3   Refining the implementation of the core

This section presents our results of applying the tools to refine and optimize the contents of some of the large classes in the Android core environment.

For this example we looked at the `TextView` class that represents a widget displaying text. All widgets such as the `Button` and `EditText` objects, extend the `TextView` class since they also display text. The `mText` field declared in the `TextView` class is used to store this text internally. This is important for applications like the Calculator application where the text on the buttons are used in the logic of application implementation (Refer to Listing 2). Since the `mText` field is an important field in the `TextView` class, we ran OCSEGen to identify which methods reference this field. Listing 4 below presents two methods which side-effect analysis could clearly identify.

Analyzing this class using OCSEGen shows us: (1) which methods have no side-effects on this field and can be stubbed and (2) which methods have effects on this field, such as the `getText` and `setText` methods, and have to be inspected.

To preserve all references to the `mText` field, we ran Modgen to slice the `TextView` class given only the `mText` field. Modgen reduced the `TextView` class from 9476 to 5827 LOC but the model

```
1  public java.lang.CharSequence getText(){
2      return this.mText;
3  }
4
5  private void setText(java.lang.CharSequence param0, android.
       widget.TextView.BufferType param1, boolean param2, int
       param3){
6      java.lang.CharSequence r0 = null;
7      r0=param0;
8      if(Verify.randomBool()){
9          this.mText=r0;
10     }
11 }
```

Listing 4: Extract from the TextView Stub created by OCSEGen

that was generated was over-approximated. Its slicing algorithm kept references to all the variables and code effecting this field which resulted in the tool preserving too much of the original code in the model to be of any insight as this stage. The reduction in size can mainly be attributed to the removal of code in methods not referencing the `mText` field which can be done more effectively using OCSEGen. Modgen can also not retain call hierarchies since it only performs intra-procedural analysis on a single class at a time. The generated stub of the `setText` method, for example, was not reduced in size by the slicing and other methods that calls the `setText` method was stubbed out since the tool could not detect changes further down in the call hierarchy.

The biggest issue with using the slicer is the decompilation of the generated Java byte-code back to Java source code. We used the the Java Decompiler (JAD) [3] and Procyon [4] but both of these tools are still in development and could not completely decompile the byte-code into compilable Java source-code. This is a problem since fixing the code manually does not scale to large classes.

5. DISCUSSION
By investigating the two static analysis tools, we found that both side-effect analysis and slicing could be useful in generating environment models.

Side-effect analysis is designed to detect modifications to the fields of interest, but may not be able to detect what kind of modification was made. For example storing or removing from an array is detected as a modification, but side-effect analysis can not distinguish between them whereas slicing can. We still need to inspect each side-effect manually to identify and preserve its effect.

Slicing on the other hand is used to optimize classes for which not all behavior is relevant to the verification of the unit. Slicing has the advantage that it will preserve all functionality in the original class in a sound way by removing unused behavior. For small classes this works well but for larger, more complex classes slicing includes too much of the original implementation since all references to the specific fields are preserved in the model.

To refine the core classes we can combine these techniques to identify methods that have a side-effect on certain fields and then retain the changes by looking at a sliced version of the class. The main problem is the capabilities of the slicer.

There are three improvements to the tools that can be investigated. The first improvement is adding support to OCSEGen

---

to not only identify methods called from a set of classes, but to identify all callers of a specific class or method. This will enhance the driver generation capabilities of OCSEGen.

The second improvement is to extend OCSEGen to merge results of the different runs of stub generations and side-effect analysis instead of inspecting them manually.

Another improvement is to extend and refine the slicing capabilities of Modgen to generate more efficient models.

6. RELATED WORK
OSGEGen has previously been applied to Java applications making use of the Swing GUI framework [2]. Since their work was focused on analyzing interaction orderings for the applications, the environment generation is localized to modeling the environment around the code implementing the GUI. OCSEGen was used to analyze the interaction between the application code and the Swing library to identify classes in the framework reachable from the application and preserve side-effects to the state of the GUI objects. They defined the state of GUI objects as enabledness, visibility, containment and listener registration. In this approach, the stub environment is modeled closely to the application and for the application to run, the driver needs to be smart in the sense that it needs to generate valid sequences of input events. A disadvantage of this approach is that it generates application specific stubs that need to be expanded by running OCSEGen on multiple applications to create more general stubs.

7. CONCLUSION
There are many technique available for environment generation, but it is hard to know when and where to apply these techniques to a specific domain. This research is aimed towards finding where these techniques can be applied to the Android domain to generate the environment model.

In this paper we discussed two specific static analysis tools, OCSEGen and Modgen and how they can be applied to the Android domain to improve the current approach to create environment stubs by hand. But, generation of environment is only a small part of the work that needs to be done to generate an Android environment model. Future work will investigate other techniques such as runtime monitoring and also find ways to automatically generate other parts of the environment such as the driver.

8. REFERENCES
[1] M. Ceccarello and O. Tkachuk. Automated Generation of Model Classes for Java PathFinder. In *ACM SIGSOFT Software Engineering Notes*, 2013.
[2] M. Dwyer, Robby, O. Tkachuk, and W. Visser. Analyzing interaction orderings with model checking. *Proceedings. 19th International Conference on Automated Software Engineering, 2004.*, pages 154–163.
[3] 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.
[4] H. van der Merwe, B. van der Merwe, and W. Visser. Verifying Android applications using Java PathFinder. *ACM SIGSOFT Software Engineering Notes*, 37(6):1, Nov. 2012.
[5] H. van der Merwe, B. van der Merwe, and W. Visser. Execution and Property Specifications for JPF-Android. In *ACM SIGSOFT Software Engineering Notes*, 2013.
[6] 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.

---

[3] http://jd.benow.ca
[4] https://bitbucket.org/mstrobel/procyon