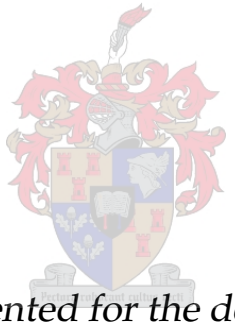
Verifying Android Applications Using Java PathFinder

by

Heila-Marié Botha



Dissertation presented for the degree of Doctor of Philosophy in Computer Science in the Faculty of Science at Stellenbosch University

Supervisors: Prof. W Visser and Prof. AB van der Merwe

December 2017

Stellenbosch University https://scholar.sun.ac.za

Declaration

By submitting this dissertation electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: December 2017 .....

Copyright © 2017 Stellenbosch University
All rights reserved.

Stellenbosch University https://scholar.sun.ac.za

Abstract

Verifying Android Applications Using Java PathFinder

H. Botha

Department of Computer Science
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa.

Dissertation: PhD (Computer Science)

November 2017

Current dynamic analysis tools for Android applications do not achieve acceptable code coverage since they can only explore a subset of the behaviors of the applications and do not have full control over the environment in which they execute. In this work model checking is used to systematically and more effectively explore application execution paths using state matching and backtracking. In particular, we extend the Java PathFinder (JPF) model checking environment for Android. We describe the difficulties one needs to overcome as well as our current approaches to handling these issues. We obtain significantly higher coverage using shorter event sequences on a representative sample of Android apps, when compared to Dynodroid and Sapienz, the current state-of-the-art dynamic analysis tools for Android applications.

Stellenbosch University https://scholar.sun.ac.za

Acknowledgments

First and foremost I would like to thank my advisors Prof. Willem Visser and Prof. Brink van der Merwe for their invaluable guidance, patience and support. Prof. Visser, thank you for teaching me to take a step backwards and look at the bigger picture, to see problems as challenges and to not be afraid to dream big. Prof. van der Merwe, you taught me to prioritize that which is important and to always ask the hard questions. I would like to thank Oksana Tkachuk for her insights and contributions to this work as well as her encouragement and enthusiasm that kept me going through the hard times.

I would like to thank the MIH Media Lab and the Center for Artificial Intelligence Research (CAIR) managed by the Meraka institute at the Council for Scientific and Industrial Research (CSIR) for awarding me bursaries which made this research possible as well as the financial support to present my work at many international conferences over the course of my research.

I would like to express my gratitude to my family, especially my parents. Their love and support were essential for the completion of my work.

Finally, I would like to thank my husband, Christoff, for his patience and loving support. It is to him that I dedicate this thesis.

Stellenbosch University https://scholar.sun.ac.za

Contents

Stellenbosch University https://scholar.sun.ac.za

Stellenbosch University https://scholar.sun.ac.za

Stellenbosch University https://scholar.sun.ac.za

List of Figures

Stellenbosch University https://scholar.sun.ac.za

Stellenbosch University https://scholar.sun.ac.za

List of Tables

Stellenbosch University https://scholar.sun.ac.za

Acronyms

API Application Programming Interface

AWT Abstract Window Toolkit

DEX Dalvik Executable

DFA Deterministic Finite Automata

DVM Dalvik Virtual Machine

GUI Graphical User Interface

IPC Interprocess Communication

JPF Java PathFinder

JVM Java Virtual Machine

LTL Linear Temporal Logic

MoP Monitor Oriented Programming

OS Operating System

SDK Standard Development Kit

SPF Symbolic PathFinder

SUT System-Under-Test

UI User Interface

Stellenbosch University https://scholar.sun.ac.za

Chapter 1

Introduction

Android applications (apps) are used for banking, shopping and accessing/storing personal information. They contain bugs and errors like all software applications, but since they operate in a safety critical environment, errors and bugs can have serious effects. Therefore, these apps need to be thoroughly tested and analyzed.

Android applications can be tested manually on a device/emulator. In this case a user interacts with the device by tapping on the screen, pressing buttons and changing the settings or configurations of the device to expose hidden or unexpected erroneous behavior of the application. Manual testing is expensive and not scalable since it relies on human testers. Additionally, applications must be tested on many different devices, device configurations and Android versions to ensure reliability. Without utilizing record-and-replay functionality, one cannot ensure the application still works after fixing a bug or for previous/future releases.

The process of manual testing is automated using dynamic analysis tools. They perform an under-approximated analysis of the application's behavior because only a subset of possible event sequences and device configurations are tested. The most basic dynamic analysis tools for Android typically exercise the application running on a device/emulator by blindly firing events (monkey testing) [20]. Random and heuristic event generation strategies are then used to optimize the analysis [17, 19,

Stellenbosch University https://scholar.sun.ac.za

22, 52, 54, 55, 57]. In order to bound the analysis, heuristics such as event sequence length or runtime are used. The effectiveness of these tools are widely measured using statement code coverage. Statement coverage measures the percentage of the program statements executed during the analysis and gives an indication of how thoroughly the code was tested.

Although these tools obtain high coverage for certain apps, they struggle to achieve sufficient coverage for applications critically dependent on their environment for large parts of the application code [28]. The environment of the System-Under-Test (SUT) consists of the application dependencies without which an application cannot run and an event generator to drive its execution. More specifically, there are two main challenges dynamic analysis tools face:

Environment Configuration Environment configurations include the battery level, network state, apps installed on the device, database and content providers' state, the file system contents and even the state of a remote web server with which an app communicates. Android only exposes limited functionality to configure its environment. Tools struggle to detect configurations used by applications and at which point to change the configuration to effectively explore the maximum number of application paths.

Event Generation Android applications require particular events at specific points in their execution to enable certain application code, for example, firing an incoming call from an exact number when the service is running. Detecting these events is hard since they might be hidden in the implementation. Apps also require specific event sequences to reach certain areas in the application code. Detecting valid event sequences and reducing them to a bounded, representative set is also a non-trivial problem due to the influence of environment configurations and the limited access to the state of the application and its environment provided to tools.

Stellenbosch University https://scholar.sun.ac.za

In this work we apply model checking to improve the effectiveness (measured using code coverage) and efficiency (measured using number and length of event sequences) of current dynamic analysis tools. Model checking is a mature verification technique used to prove the absence of property violations in a system with a finite (and small enough) state space. However, it is often used as a bug-finding tool. It systematically performs a depth-first search (or breath-first search) over the state space of the SUT represented as a graph. The search is optimized using state matching which ends exploration of a path when a previously seen state is reached, and backtracking which uses caching of previously explored states to continue the search from a branch point without having to re-execute the path leading up to the state. Model checking also provides fine-grained control over the application's execution to verify property specifications.

Software applications are model checked using program model checkers such as Java PathFinder (JPF) [63]. JPF is a powerful and established Java application model checker and analysis engine. Since Android applications are written in Java, we investigate how JPF can be applied to analyze Android applications. JPF provides mechanisms to override and control inflexible environment behavior. This enables coverage of hard-to-reach areas in the application code current Android analysis tools struggle to reach due to the application's reliance on an external environment. State matching and backtracking capabilities allow effective exploration of shorter and more influential event sequences. It also provides a property listener framework to monitor the execution of an application at byte-code level. Furthermore, if a property violation is reached, it can trace back the execution leading to the violation. JPF has many extensions that can be applied to Android applications (such as Symbolic PathFinder) once they run on JPF.

Our extension to JPF is called JPF-Android (Figure 1.1.) Since Android applications are executed instead of analyzed statically, the tool faces challenges similar to other dynamic analysis tools. To model check Android applications extensive environment modeling of missing, too

Stellenbosch University https://scholar.sun.ac.za

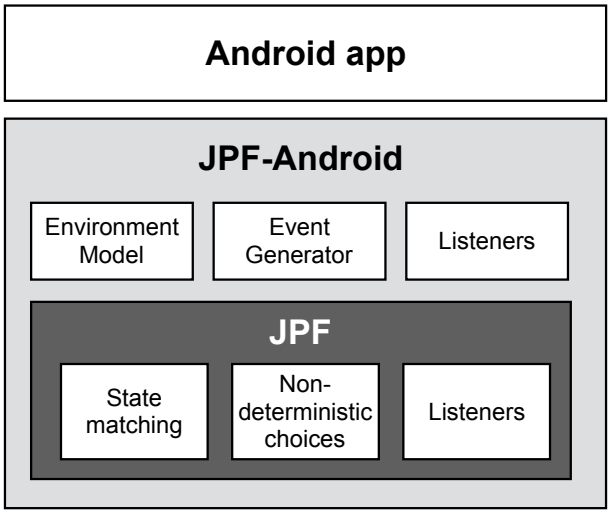


Figure 1.1: Overview of JPF-Android

large and unavailable dependencies is required. JPF-Android provides an *environment model* (Chapter 3) implementing the functionality required by applications to run. This model is based on an abstracted version of the actual application framework and provides full control over the environment behavior. The environment model also includes generated application specific models returning default/runtime/statically collected values to improve coverage. Although creating such a model requires a lot of effort, it provides full control over the environment configuration and event generation that is not possible using dynamic analysis tools.

To generate valid events and event sequences to obtain good coverage of the application code, JPF-Android provides an *event generator* (Section 4) that detects enabled entry-points and generates events to drive the execution of the application.

Although model checking optimizes the search with state matching and backtracking, it suffers from the state-space explosion problem. Additionally, Java as well as Android applications are not finite state by design. To reduce the environment models of JPF-Android, we implemented a tool to detect unbounded variables in the SUT. We also implemented ways to optimize state matching and reduce the number of branches in the state-transition graph to obtain a more manageable analysis size.

Stellenbosch University https://scholar.sun.ac.za

Lastly, JPF-Android includes a set of listeners to record the coverage, event sequences and environment configurations explored during the analysis (Section 5). To evaluate the effectiveness and efficiency of our approach we ran the tool on a set of representative apps and showed a significant increase in the code coverage using shorter event sequences in comparison to Dynodroid [52] and Sapienz [55], the current state-of-the-art dynamic analysis tools (Section 6).

1.1 Motivating Example

The two main challenges for dynamic analysis tools causing low statement coverage for Android applications are environment configuration and event generation. These challenges can be illustrated using an app from the Google Play Store: *AutoAnswer* [4]. AutoAnswer automatically answers incoming calls in certain configured scenarios. It is enabled in the app's main preference screen and can be configured to answer calls from all numbers, only contacts or only starred contacts. Additionally, a delay can be set before answering a call and whether calls should be answered using the speaker or a Bluetooth headset. Lastly, the user can configure it to answer a second incoming call.

To get notified of incoming calls the application registers a Broadcast Receiver (BR) (shown in Listing 1.1) for `PHONE_STATE` events. If the phone-state changes to `STATE_RINGING` and the service is enabled (lines 10 – 11), the contact restrictions are checked (lines 15 – 22). If the call should be answered, the `AutoAnswerIntentService` is started to answer the call (line 25).

1.1.1 Environment Configuration

Dynamic analysis tools have limited control over the environment of the application on an Android device. They can miss important behavior because relevant configurations cannot be detected, specific environment

Stellenbosch University https://scholar.sun.ac.za


```
1 public class AutoAnswerReceiver extends BroadcastReceiver {
2   @Override
3   public void onReceive(Context context, Intent intent) {
4     // Load preferences
5     SharedPreferences prefs = PreferenceManager.
      getDefaultSharedPreferences(context);
6
7     // Check phone state
8     String phone_state = intent.getStringExtra(TelephonyManager.
      EXTRA_STATE);
9     String number = intent.getStringExtra(TelephonyManager.
      EXTRA_INCOMING_NUMBER);
10    if (phone_state.equals(TelephonyManager.EXTRA_STATE_RINGING)
11        && prefs.getBoolean("enabled", false)) {
12      ...
13      // Check for contact restrictions
14      String which_contacts = prefs.getString("which_contacts", "
      all");
15      if (!which_contacts.equals("all")) {
16        int is_starred = isStarred(context, number);
17        if (which_contacts.equals("contacts") && is_starred < 0) {
18          return;
19        } else if (which_contacts.equals("starred") && is_starred <
       1) {
20          return;
21        }
22      }
23
24      // Call a service, since this could take a few seconds
25      context.startService(new Intent(context,
      AutoAnswerIntentService.class));
26    }
27    ...
28  }
```

Listing 1.1: Code extract from AutoAnswerReceiver

configurations cannot be set or when there are too many configurations to consider.

The AutoAnswer application depends on external services/libraries including the contacts Content Provider (CP) that allows the application to query the list of contacts on the phone, the Bluetooth service used to check if a headset is connected and the audio manager used to play audio through the speaker. Configuring these environment dependencies can be challenging. For the application to exercise its behavior related to the Bluetooth service, for example, we need to run the application while physically connecting and disconnecting a Bluetooth headset. On

Stellenbosch University https://scholar.sun.ac.za

the emulator this is not even possible since Bluetooth support is not emulated.

The return values of these dependencies directly influence the application's code coverage but are often hidden in the implementation of the application. The contacts CP, for example, should contain starred and not-starred contacts matching the incoming call's number to enable lines 15 – 22 in Listing 1.1.

Dependencies can return many different values but the input space should be limited to representative return values. The application only responds to two values returned from the Bluetooth service: headset connected or headset not connected. If the Bluetooth service returns any other value, it will not enable any new application behavior.

1.1.2 Event Generation

Another challenge for dynamic analysis tools is generating event sequences for Android applications because of the many possible events and event combinations (especially when combined with different environment configurations). The events that can be fired for AutoAnswer include changing each of the settings on the preference screen as well as firing the `onReceive()` method of the BR with `PHONE_STATE` events. Dynamic tools cannot detect that an incoming call event should be fired for each environment configuration (all preference combinations). They might fire the BR multiple times for the same configuration, but miss firing it for important configurations or stop analysis before all configurations have been explored.

Secondly, the parameters with which entry-points are called, have a big influence on the application's code coverage. Although the event generator can detect that the BR is registered for `PHONE_STATE` events, there are many different such events: `STATE_RINGING`, `STATE_PHONE_OFFHOOK` and `STATE_PHONE_IDLE`. Each of these events have different parameters. The `STATE_RINGING` event, for example, has an `EXTRA_INCOMING_NUMBER` parameter (lines 8 – 9). In this example this param-

Stellenbosch University https://scholar.sun.ac.za

eter is crucial. If the number is null, the app throws a `NullPointerException` in the `isStarred()` method. To enable lines 16 – 21 in Listing 1.1, the incoming number must match a contact, a starred contact and no contacts.

1.2 Contributions

The contributions of this work include:

1. **Environment modeling of missing, too large and unavailable/unconfigurable dependencies**
    - Identification of what components need to be modeled
    - Investigation of available tools to automate modeling
    - Identification of where tools can be applied for modeling the Android environment
    - Development of an Android environment model and modeling strategy
2. **Generating valid events and short, bounded and representative event sequences**
    - Design and create driver based on Android event-driven message-passing design to collect, and non-deterministically fire events
    - Improvement of coverage by extension of driver to automatically use runtime/statically collected parameters for entry-points
    - Provide mechanism to implement custom event generation strategies
    - Implementation of random, dynamic and heuristic event generators
    - Tracking of explored event sequences using an event tree

Stellenbosch University https://scholar.sun.ac.za

3. **Optimizing model checking to enable analysis of Android applications**

    - Detection of unbounded variables
    - Reduced number of classes in state
    - Pre-load all application classes
    - Customize exploration of non-deterministic choices
    - Track and verify execution per thread using Checklist

4. **Evaluation of the effectiveness of these techniques and a comparison to state-of-the-art dynamic analysis tools**

1.3 Overview

The thesis is divided into six chapters. Following Chapter 1, the introduction, Chapter 2 provides background information on the Android environment and describes model checking as a verification technique. It then continues to discuss related work. Chapter 3 outlines the challenges of environment modeling for Android applications and discusses our approaches to overcome these challenges. Chapter 4 explains how events are generated to drive the execution of the Android application under analysis. Chapter 5 describes the optimizations and extensions made to JPF in order to effectively analyze Android applications. The tool is evaluated in Chapter 6 by comparing it to state-of-the-art dynamic analysis tools: Dynodroid and Sapienz. Chapter 7 concludes the thesis by providing a summary of the research and suggesting future refinements and extensions to the tool.

Stellenbosch University https://scholar.sun.ac.za

Chapter 2

Background

This chapter gives background information on Android application design and model checking followed by a discussion on related work. Firstly, the Android software stack and Android application internals are described to understand application behavior. Section 2.2 gives an overview of model checking and the capabilities of Java PathFinder (JPF). Next, approaches to environment modeling are given before concluding with a discussion on current Android application analysis tools and their strengths and weaknesses.

2.1 The Android Software Stack

Android is an open source *software stack* (Figure 2.1) developed to run on mobile devices with resource and energy constraints. It is designed to support a range of device configurations and capabilities [31].

The Android software stack is built on top of a modified Linux kernel and includes modules such as the Binder Interprocess Communication (IPC) driver and a power manager to satisfy the specific needs of mobile devices. The kernel is responsible for efficient memory, process, user and thread management. It includes and supports several configurable native libraries and drivers such as the web browser engine (webkit), libc library, camera driver, audio manager, media framework, database engine and

Stellenbosch University https://scholar.sun.ac.za

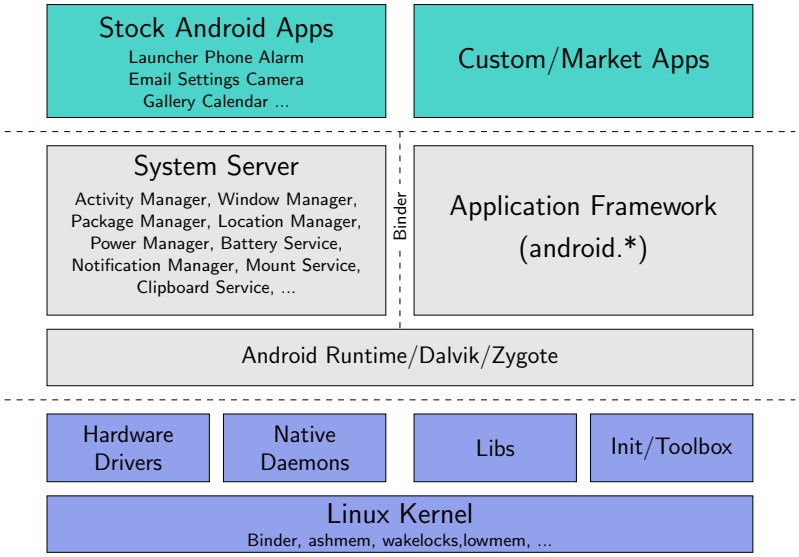


Figure 2.1: The Android software stack

the graphics engine [18]. For security reasons, each Android application runs as a separate process. [1] This allows Linux's built-in process and user management to keep applications from modifying the data and code of other applications.

Android applications, the system services as well as the application framework that provides the base implementation for applications are implemented in Java. The application framework simplifies application implementation and communication with native libraries, drivers, other applications and external services. The Java code is compiled using the standard Java compiler and converted to a compacted, optimized version of the Java byte-code called Dalvik Executable (DEX) byte-code. DEX is executed on the Dalvik Virtual Machine (DVM) which runs in the application's process. The DVM resembles the Java Virtual Machine (JVM), but has a register-based design compared to the JVM's stack-based design. The DVM provides an abstract environment and hides the complexity and variability (inconsistency) of the environment on a device.

System services run in their own process and perform Operating Sys-

[1] There are ways to run multiple applications in the same process, but this is not common and will not be discussed here.

Stellenbosch University https://scholar.sun.ac.za

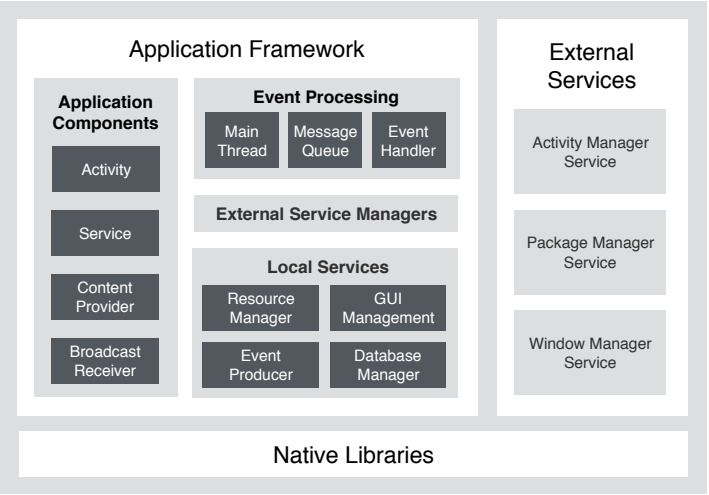


Figure 2.2: The Android application environment

tem (OS) level tasks. There are three main services shared by all applications. The `ActivityManager` service is responsible for managing the state and interaction between application components across all running applications. The `PackageManager` service keeps information on installed packages. The `WindowManager` service stores the window currently on the screen and maps Graphical User Interface (GUI) events to the correct application and listener.

At the top of the stack is the stock Android applications shipped with the Android platform as well as custom Android applications created by third party vendors.

2.1.1 Android Applications

Android applications consist of a collection of application components that run on top of the extensive Android application framework (Figure 2.2.) The framework provides four base components that can be extended to implement a basic application. An *Activity* is used to instantiate and control a GUI, a *Service* performs background tasks, a *Broadcast Receiver (BR)* is used to subscribe to specific events and lastly a *Content Provider (CP)* manages data access. These components provide specific callback methods fired by the framework to transition them between states. The

Stellenbosch University https://scholar.sun.ac.za

life-cycle of each component defines the different states of the component and the sequence of callback methods called for each transition. These base components are extended by Android applications and their callback methods are overwritten in order to extend the behavior of a component.

Android applications have a single-threaded event-based design. Each application has one main event handling thread that drives its execution by processing messages (containing events and callbacks) on the main message queue of the application [15]. This design is commonly used by GUI frameworks since it is too complex and inefficient to make all User Interface (UI) classes thread safe.

Application entry-points include listeners and callbacks registered in the application framework. These callbacks are fired by events. Events can be triggered by user interaction with the application such as clicking a button on the UI, key presses on the keyboard, as well as physical button presses such as the volume, back or home button. Events are also triggered by the environment due to a change in the state of the underlying system. This includes the location changing, battery level notifications, network connectivity changes and alarms. Life-cycle methods of the application components are called implicitly by the framework.

Due to the fact that Android applications have a single main thread to handle all application events, most applications use some form of concurrency to avoid the main thread from blocking and becoming unresponsive. All long running operations, including common operations such as network and database operations, are required to execute in a new thread. Android introduces the `AsyncThread` object as a way to simplify concurrency. The `AsyncThread` object stores a static thread pool of Java threads per application. It is used to schedule tasks that need to execute in a new thread. The `AsyncThread` object provides the functionality to schedule messages on the message queue of the main thread to provide feedback on the status of the task.

Android application components interact asynchronously with other components and external services by using Android's Binder IPC mechanism to pass messages. These messages are called `Intents`. `Intents`

Stellenbosch University https://scholar.sun.ac.za

are used to notify application components of certain events. They can contain a description of an operation to be performed or, in the case of broadcasts, a description of something that has happened and is being announced. Application components follow the observer design pattern by registering for specific broadcast messages. `IntentFilters` are used to register components for specific types of events to be forwarded to a component.

Applications do not have direct access to external services but instead use locally running service managers to facilitate the IPC. These managers are responsible for serializing method calls, parameters and return values and storing and firing callbacks of the application in response to the remote service's behavior.

Local services are implemented as part of the application framework and run in the application process. They are used to access local files and data and to give access to native libraries. These services include the `LayoutManager`, `ResourceManager` and camera.

2.2 Model Checking

Model checking is a formal verification technique that automatically proves the correctness of a System-Under-Test (SUT), given a logical formula describing a property specification [29]. Correctness is determined by extracting a finite-state, directed, state-transition graph from the SUT and performing a path-sensitive analysis of the graph to determine if it satisfies the logical formula (i.e., determine if the graph is a model of the formula) [29, 44]. A path-sensitive analysis tracks variable values and evaluates conditional expressions to only explore valid, feasible program paths. If a formula is not satisfied, the model checking engine can trace back the path in the state-transition graph as a counterexample leading to the violation.

These logical formula specifications can describe safety properties such as unchecked exceptions and liveness properties. They enable model checking to search for and detect concurrency errors in multi-threaded

Stellenbosch University https://scholar.sun.ac.za

programs which are difficult to detect using program testing [29] such as deadlock and race-conditions. Property specifications can also be used to verify that the implementation conforms to its design requirements.

2.2.1 Explicit-State Model Checking

Explicit-state model checking generates states explicitly, on-the-fly while performing a depth-first search of the state-transition graph (but it can also perform a breadth-first search). Its goal is to find property violations. When the search ends without finding a property violation in a finite-state system where all states are explored, the system does not violate the property.

Each *path* in the state-transition graph represents a possible execution of the application. A *state* represents the state of the SUT and a *transition* represents the list of program instructions executed between two states. The SUT is executed deterministically until multiple paths are possible due to different *choices* in the environment. These choices include, for example, different user inputs, thread schedulings or random number values [44]. At these choice points the transition is ended and the current state of the system stored. The different choices are explored non-deterministically, originating from the same saved state. Backtracking enables model checking to restore a previously saved state to explore all of its non-deterministic transitions without having to re-execute the path leading to the saved state. Exploration is stopped when an end state (termination of the program) or a previously visited state (in which case the following execution was already explored) is reached.

Explicit-state model checking is exponential in the size of the graph. Therefore, state matching stores and matches states to previously visited states. This ensures that each state is only explored once — the first time it is reached — reducing the search to be linear in the size of the graph.

The SPIN model checker [40] is an example of an automata-theoretic explicit-state model checker that uses Linear Temporal Logic (LTL) formulas to describe property specifications. These formulas are then converted

Stellenbosch University https://scholar.sun.ac.za

to Büchi automata which accept if the execution does not violate the specific property specification. In this case the search is linear in the number of states, but exponential in the number of properties.

2.2.2 Java PathFinder (JPF)

JPF is a mature, open-source, analysis engine for Java applications [63]. It is implemented as an explicit-state model checker that works directly on Java class files. JPF provides a custom, abstracted JVM that is implemented in Java and runs on the Oracle JVM. It includes callbacks to notify subscribed listeners of certain JVM and JPF events. The application is executed on JPF while being verified against property specifications implemented as JPF listeners. By default, JPF provides listeners to detect errors such as deadlocks, race conditions, infinite loops, track object allocations and can inject exceptions into the application.

A *transition* in JPF consists of the list of byte-code instructions leading from one state to another. A *state* consists of the values of all variables in the system and includes [48] a:

Thread List Stores a list of the threads and their current states. The state includes a thread's unique id and stack trace. The stack trace contains a list of stack frames — one for each method call storing the location in the method (program counter), its local variables and operands.

Static Area Stores a list of all loaded classes and the values of their static variables.

Dynamic Area Stores a list of all reference (dynamic) objects in the heap. Each entry contains the type of the object, its unique reference id and a map of its fields and/or list items and their values. Reference ids represent memory locations in the heap and are stored as integer values.

Variables can either store primitive or complex data types. Primitive data types include `boolean`, `integer` and `char`, for example, and store

Stellenbosch University https://scholar.sun.ac.za

their value directly in the variable. Complex data types are a composite of primitive and other complex data types. Variables storing complex data types only store a reference to the actual object in the dynamic area.

When a new state is saved by JPF, the current thread list, classes (static area) and objects in the heap (dynamic area) are serialized into a hash value. To retain heap symmetry, object entries are serialized in the order in which they were created. This hash value is then used to detect state matches by comparing it to previously visited states' hash values. To improve state matching, garbage collection is run before a state is stored to filter out unreferenced objects.

Java applications are not finite state by design because they depend on a (possibly) unbounded environment. We can create a finite-state system by abstracting environment behavior (and sometimes the application itself) and by limiting the search depth in the case where the abstraction is not enough or the application state space is unbounded. JPF allows classes to be modeled/abstracted by replacing any Java class in the application, its libraries or even the Java class library with a modeled version.

JPF includes an option for state debugging. When enabled, a serialized version of each state is stored in a file. These files can be compared by using a diff tool in order to identify changes in the state.

JPF separates the process of state matching and state storage for backtracking. The serialization of the state into a hash value is only used for state matching. In order to restore the state of the system, JPF stores a memento of the kernel state (thread list, heap, classloaders and listeners) from which the state is restored.

JPF simplifies specifying properties by providing property listeners to track byte-code execution. These listeners can be extended and implemented in Java to verify all kinds of properties including liveness and safety properties. Although JPF does not natively support specifications in the form of logical formulas, an extension to the tool, JPF-LTL provides this functionality [1].

JPF's design enables developers to easily create extensions to the framework. Currently, there exist many extensions to JPF including a

Stellenbosch University https://scholar.sun.ac.za


```
1 $imageFileNameEdit.setText("sunset.png")
2
3 // select a transformation
4 REPEAT 2 {
5   ANY { NONE, $<greyScale|resize>Button.doClick() }
6 }
```

Listing 2.1: Example of an JPF-AWT input script

symbolic execution extension (JPF-SYMBC) [58], data race detector (JPF-RACEFINDER) [45] and an Abstract Window Toolkit (AWT) extension (JPF-AWT) [56].

JPF-AWT enables the model checking of AWT applications. AWT applications, similar to Android applications, are based on a single-threaded, message queue design. All application events are put in a message queue and then processed by the main thread of the application, called the `EventDispatchThread`.

As AWT applications are event-driven, JPF-AWT introduced the idea of using a simple event script to write sequences of user inputs to drive the application execution. This is done by modeling the `EventDispatchThread` class to request events from the script file simulating the event occurring, when the message queue is empty. As model checking is usually done on a closed system, the script drives the application to execute specific functionally of the application to be verified.

Since multiple events can be fired at a specific point in time, the AWT input script introduces the Alternative (Any) script element to explore a set of event choices non-deterministically (see Listing 2.1). JPF-AWT only supports one level of non-determinism so does not support Any elements inside other Any elements.

JPF-AWT has the advantage that AWT applications make use of the Java library classes for operating system functions such as network services and file managing, whereas Android applications use custom Android libraries implemented for the Android platform.

Stellenbosch University https://scholar.sun.ac.za

2.3 Environment Modeling

In testing and analysis the SUT consists of two components: the *unit* under test and its *environment* [30]. The environment includes all libraries and components with which the unit interacts. The environment can be broken down into the *driver* and the *dependencies*. The driver executes the unit by making calls into the application code whereas the dependencies represent classes referenced from the unit. Note that in reality dependencies can contain callbacks to the application or to the driver. Models (also called mocks or stubs) are used to abstract dependencies such as external libraries or services that are unavailable, inflexible or irrelevant to the analysis. Figure 2.3 shows a simplified view of an environment model used to create a closed system, without external references, to analyze the unit in isolation.

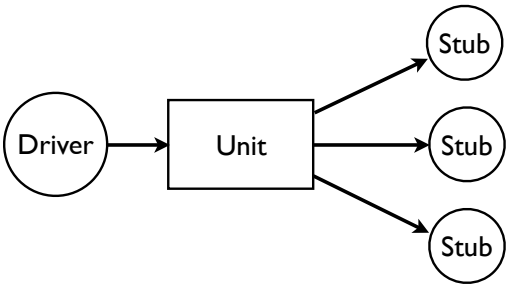


Figure 2.3: Structure of an environment model

JPF allows developers to specify an environment model to simulate the behavior of the actual environment. JPF, together with its many extensions[2] including jpf-concurrent, jpf-awt, net-io-cache, provide models for many of the Java library classes. NetIO [49] and NetStub [25], for example, 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 JPF and interact with its counterparts running

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

Stellenbosch University https://scholar.sun.ac.za

external to the verification tool. This interaction is cached to allow JPF to backtrack during analysis.

Dependency models can be created by hand, or generated automatically using static or runtime techniques. Environment generation is aimed towards the automatic generation of environment models. Static analysis tools geared towards environment generation such as OCSEGen and ModGen produce over-approximated models whereas runtime analysis tools under-approximate the behavior being modeled.

OCSEGen [61] can generate both drivers and dependency models for the unit. The driver generation is configured by user specifications given in LTL or regular expressions. For model generation the tool makes use of side-effect analysis. The tool has many configurations for model 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 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 used to generate code for models.

The tool 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 model 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 models when the application is run on a symbolic execution tool such as Symbolic PathFinder (SPF) [58]. OCSEGen has limitations typical to static analysis: it may produce over-approximation of side-effects and cannot analyze native code or code using reflection.

Stellenbosch University https://scholar.sun.ac.za

OSGEGen has been applied to Java applications making use of the Swing GUI framework [30]. Since the 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. Side-effects to the state of the GUI are then preserved. The authors define the state of GUI components as enabledness, visibility, containment and listener registration. For the application to run, the driver also 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 manually by running OCSEGen on multiple applications to create more general stubs.

Modgen [26] 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. 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.

Another tool, nHandler [60], automatically models methods of libraries/applications on JPF. It delegates the unavailable method's execution to an instance of the object in its actual environment on the JVM and then converts and returns the result from the modeled method in the JPF environment. This approach assumes that the method and the library it belongs to can be instantiated and run directly on the JVM. This is not the case for Android applications since their external libraries have

Stellenbosch University https://scholar.sun.ac.za

dependencies on native services and services not available outside of the Android environment.

Work has been done to model external code for static analysis using data flow summaries [23]. The requirements for models used for static analysis are less complex since the application code is not executed.

The Android Standard Development Kit (SDK) includes a few models for unit testing and mocking frameworks, such as Mockito [13], can generate models for classes/methods referenced from the unit. These models return default values or can be extended manually to return expected values. For unit and functional testing smaller parts of the application are tested such as a method or component. The models are therefore smaller and less complex. To enable non-determinism, state matching, component interaction and correct life-cycle management of components, we require a more complex model of the environment.

Recent work [43] has started to use design pattern recognition to classify and automatically model dependencies using predefined implementation of these patterns. Although very promising work, we found that in a large framework such as Android, design patterns are blurred and dependencies and object hierarchies severely complicate automatic modeling.

2.4 Runtime Verification

Runtime verification verifies the correctness of an application by monitoring an execution trace of the application and comparing it to user-defined property specifications. These property specifications are typically described using formalisms such as LTL and extended regular expressions. Techniques such as logic-based monitoring and error pattern analysis are then used to verify the application's execution trace against these specifications to detect violations of safety and liveness properties [27, 46, 39].

Runtime verification is designed to fill the gap between formal verification methods and testing. It avoids the complexity of formal methods such as model checking and theorem proving, but utilizes their temporal

Stellenbosch University https://scholar.sun.ac.za

logic specifications to describe the execution of the application. Testing on the other hand is very scalable. Runtime verification merges the expressiveness of logic specifications with the scalability of testing [39].

Runtime verification allows applications to be monitored either *inline* or *offline*. Inline monitoring requires modification of the application's code to insert annotations/comments describing the property specifications of the program. During a precompilation stage, these comments/annotations are used to dynamically generate code verifying the properties. The disadvantage of using inline monitoring is that the monitoring operations influence the application's execution. Offline monitoring entails instrumenting the code/byte-code of the application to emit events indicating changes in the program state. These event traces are processed by an external system, independent of the application, to detect errors.

Chen and Roşu [27] developed a Monitor Oriented Programming (MoP) paradigm to verify that application execution conforms to logic based specifications. Their system aims at providing a language and logic independent solution to runtime verification and to provide both inline and offline monitoring. They implemented an environment to verify Java applications using past time and future time LTL as well as extended regular expressions by using logic plug-ins to their system.

Runtime Verification is often used in mobile application analysis [64, 54]. Applications are instrumented to print logs while running on a device. These logs are then monitored for exceptions or other properties by an external component. The advantages of this approach include that it is mostly platform independent, does not require modeling of the application environment, and is scalable.

2.5 Android Analysis Tools

There is a whole body of work using static analysis for detecting security and privacy violations in Android applications [24, 32, 51, 33, 50]. Two drawbacks of using static analysis are that it reports false positives and does not provide an event sequence or environment configuration

Stellenbosch University https://scholar.sun.ac.za

to dynamically verify that errors exist. It also requires modeling of unavailable and native code. Our work, however, focuses on performing path-sensitive dynamic analysis.

Roboelectric [16] and JUnit testing make use of mocks to test the unit or functionality locally — outside of a device. Tools such as UIAutomator [19], Espresso [9], Robotium [5], Calabash [8], Appium [7] and the Android Instrumented JUnit framework [17] are dynamic analysis tools that allow the user to write tests executed on the emulator. Running them on the emulator reduces modeling since the applications run in their actual environment.

Automated dynamic analysis tools generate events to drive the execution of the application. Monkey [20], for example, is a random testing tool shipped with Android and fires random events to exercise application code. It detects errors in the form of exceptions. Dynodroid [52] is built on the Monkey Application Programming Interface (API) and focuses on improving coverage by using a heuristic event generation approach. Sapienz [55] makes use of multiple emulators to generate event sequences and identifies an optimal set of sequences using the Pareto-optimal genetic search algorithm to maximize the coverage and minimize event sequences.

Tools such as Evodroid [53] and Trimdroid [57] generate Android JUnit or Robotium [5] test cases to exercise the application code. Trimdroid [57] minimizes test sequences statically using dependency analysis between event handlers to reduce test cases. In contrast, JPF-Android implicitly performs dependency analysis using state matching and supports system events and environment configuration not supported by Trimdroid.

Other tools analyze logs generated by instrumented applications running on the emulator to identify bugs [41], resource leaks or race conditions [54].

All of these tools require environment modeling to some degree to achieve satisfactory coverage. This may include modeling an HTTP connection, connecting to a remote service or simulating a certain environment configuration. They also need to generate event sequences and

Stellenbosch University https://scholar.sun.ac.za

specific event parameters to improve code coverage. Dynamic analysis tools need to be run several times to obtain sufficient results.

Research has been done on collecting events at runtime and building GUI models that can be used to generate valid event sequences for Android applications [65]. The main problem these tools face is that Android applications' entry-points are enabled and disabled dynamically as the application runs which makes it hard to determine valid sequences. They also face the problem that applications respond to events that can be fired at any time while the application is running resulting in complex models.

GreenDroid [67] makes use of JPF to detect energy problems by tracking API usages. It models dependencies manually and randomly generates events to fire entry-points. The code coverage they obtain is low – less than 39% for all but one small application. JPF-Android focuses on improving coverage for Android applications while reducing the search space. The techniques we employ to optimize our tool could easily by used to improve its effectiveness and efficiency of this tool.

More advanced techniques such as symbolic execution [58] is computationally very expensive and require more extensive modeling. The complexity increases as the number of symbolic variables and number of paths increase. For this reason the analysis is optimized for GUI and event-driven applications which have many different program behaviors: by reducing the number of possible user/environment inputs [66, 42], randomly choosing a thread schedule [59], bounding the depth of event sequences [34, 22] or analyzing event handlers in isolation [35]. Note that these optimizations can have a negative impact on the code coverage achieved by the tool. Symbolic analysis tools also struggle to analyze behavior dependent on complex objects or that performs complex computations.

2.6 Summary

This chapter started with a discussion on the complexities of the Android software stack and the internals of the Android platform. Next, model

Stellenbosch University https://scholar.sun.ac.za

checking and the advantages and challenges to this verification technique were presented — focusing in particularly on JPF, a model checker for Java applications. Environment modeling was then introduced followed by a discussion on the current event generation techniques and tools available. An overview of runtime verification followed, highlighting its similarities to model checking. Lastly, an overview of current Android analysis tools was given describing their advantages and shortcomings. In the next chapter, environment modeling for Android applications is discussed including the challenges faced by JPF-Android in this regard and the approaches used to create an environment model.

Stellenbosch University https://scholar.sun.ac.za

Chapter 3

Environment Modeling

Android applications have many dependencies. In order to run them on the Java PathFinder (JPF) Java Virtual Machine (JVM), broken, unavailable and complex dependencies are modeled to form an abstracted, closed environment model. In this chapter we start by presenting our modeling approach. Next we describe the different types of dependencies of Android applications as well as the tools used to create an environment model suitable for model checking.

3.1 Overview

As discussed in Chapter 2, the System-Under-Test (SUT) can be divided into the *unit* and its environment consisting of a *driver* and *dependencies* (see Figure 2.3). For this project, the unit is an Android application. The driver fires events to trigger the entry-points of the application and is discussed further in Chapter 4. The dependencies include classes referenced from the application and components required to properly execute the application.

Dependency modeling for model checking has three main goals: (1) create a closed system by modeling unavailable dependencies, (2) abstract the original, complex environment to reduce analysis size and (3) model inflexible dependencies to improve the code coverage.

Modeling dependencies is a complex process. They can implement

Stellenbosch University https://scholar.sun.ac.za

complex class hierarchies, design patterns and can be interdependent. Dependencies can also behave differently for different environment configurations and different applications. Therefore, modeling is usually done manually. For manual modeling, the application and its environment are inspected manually and models created by hand. Before modeling a class, its dependencies and related class hierarchies need to be understood. Dependencies have references to other parts of the system and native libraries without which they cannot run or be set up correctly. The methods and fields of a dependency, referenced from applications, should then be identified and modeled. A class is modeled by abstracting its original behavior. The model should reduce dependencies on other external components and only include behavior essential to running an application.

Manual modeling can be automated using environment generation techniques. These techniques use static analysis to automatically identify all classes referenced from a unit and then generate models for these dependencies.

There are two main strategies for modeling class dependencies. A *complete stub* models all public methods of a class and can be reused by many apps. An *application specific stub* only includes methods and fields of a class referenced from a specific application. They are more concise, but cannot usually be reused by other applications.

These simplest way to generate implementations for these stubs is to return default values from their methods. For example if a method returns a boolean, it is modeled to always return "false". These models are called *empty-stubs*. Empty-stubs are simple to generate, but do not always achieve good coverage results. Instead of returning default values, symbolic execution can be used to detect application entry-point parameters or dependency return values, but cannot always be run due to the complex nature of the objects involved. To improve on the default values returned from empty-stubs, we use runtime collected values. Returning specific values from dependencies works well in cases where the application depends on specific values and dependencies have no side-effects

Stellenbosch University https://scholar.sun.ac.za

the application itself.

In some cases, however, side effects to fields in the application or its environment must be retained for proper execution of the application. These fields include listeners registered in the environment to be fired for certain events, the hierarchy of GUI elements passed to the window manager and the state of the media player. Side-effect analysis can detect and retain modifications to fields of interest in the application and its environment. It generates a call graph for the system and then performs inter-procedural analysis of methods to detect side-effects to these fields. Object references are tracked using points-to analysis and method summaries are generated retaining updates to these fields. Relevant fields are selected manually by mapping them to properties the application requires to function properly.

Slicing is a modeling technique used to optimize classes for which not all behavior is relevant to the verification of the application. Slicing has the advantage that it preserves all functionality in the original dependency in a sound way by removing unused behavior. For small classes this works well but for larger, more complex components, slicing usually includes too much of the original implementation.

For this work we combine these approaches to build on each other's strengths.

3.2 The Android Application Environment

Android applications are notoriously difficult to test and verify due to their many dependencies. There are four main dependency types in the Android environment:

The Application Framework The base implementation of Android applications on which they are heavily dependent. This code runs as part of the application process.

External System Services Operating system services running in their own process controlling the global device state for example the

Stellenbosch University https://scholar.sun.ac.za

`ActivityManagerService` and `WindowManagerService`.

Libraries Native and Java libraries referenced from the application for example Gson and commons-io.

Other Android Applications Other applications installed on the device for example the camera or browser applications.

Model checking Android applications outside of their original environment on an Android device and on JPF presents the following challenges:

Native Code The application framework depends on many native libraries and drivers. These native libraries are not available and cannot directly run outside of the emulator since available drivers, kernel modules, CPU architecture and hardware differ. This results in broken native methods in framework classes and affects commonly used XML resource parsers (used for Graphical User Interface (GUI) inflation, resource parsing, preferences) and local services such as SQLiteDB and the camera. Interaction between application components and external services, facilitated by the framework, is also broken since it is implemented as a native Binder kernel module.

System Services External system services run in their own process and Dalvik Virtual Machine (DVM). Due to their complexity and size, it is infeasible to analyze their execution together with the application. The application framework depends on these services to control application state and the state of the device. Therefore, carefully constructed models are required to abstract their behavior for a single application.

Applications Android applications interact with each other using the application framework, Binder Interprocess Communication (IPC) and the `ActivityManagerService` which resolves messages to the relevant components and apps. Interaction with external applications is therefore modeled because external applications are not available and

Stellenbosch University https://scholar.sun.ac.za

the native Binder library and external system services (including the `ActivityManagerService`) are modeled.

Libraries Libraries referenced from the application are not part of the unit and should also be modeled. This is especially hard for complex libraries performing Network I/O and file operations.

3.3 Automating Modeling

In order to automate model generation we investigated two static analysis tools: OCSEGen [61] and Modgen [26]. Both tools can generate models automatically using static analysis, but OCSEGen can also detect the dependencies of a specific unit. Since dependencies are not detected automatically by Modgen we focused on OCSEGen.

OCSEGen is very efficient at generating empty-stubs. We use the stub generation capabilities of the tool to generate complete and application specific stubs. Existing models and classes reused from the framework are excluded from the analysis.

Complete stubs are generated by analyzing the dependency class itself and generating an empty-stub. These stubs are useful when the current implementation of a dependency cannot run outside of the Android software stack. It can also abstract dependencies not important to the analysis or with no side-effects on the analysis. Complete stubs are used as base implementations for service managers or local services with native code as well as for abstracting graphical or accessibility classes.

Application specific stubs are generated by statically analyzing the dependencies of the application classes. OCSEGen then generates empty-stubs for classes referenced from the application. The generated models only include methods and fields called from the application. This functionality is used to create a closed environment for a new application by modeling classes not yet modeled by JPF-Android. It is also used to create custom models that can be configured to improve coverage or reduce

Stellenbosch University https://scholar.sun.ac.za


```
1 public final class Rect implements android.os.Parcelable {
2   public int bottom;
3   public int left;
4   public int right;
5   public int top;
6   public static Rect TOP = new Rect();
7
8   public Rect() {}
9
10  public Rect(int param0, int param1, int param2, int param3) {}
11
12  @Override
13  public boolean equals(java.lang.Object param0) {
14    return Abstraction.TOP_BOOL;
15  }
16
17  @Override
18  public int hashCode() {
19    return Abstraction.TOP_INT;
20  }
21
22  @Override
23  public java.lang.String toString() {
24    return Abstraction.TOP_STRING;
25  }
26
27  public static android.graphics.Rect unflattenFromString(java.lang.
      String param0) {
28    return android.graphics.Rect.TOP;
29  }
30  ...
31 }
```

Listing 3.1: Stub of the Rect class

analysis size for a specific application. These models can be changed manually to extend their behavior for a specific application.

Figure 3.1 shows an extract of a complete stub for the `Rect.java` class. Its methods return default values configured in the Abstraction class (0 for int, false for boolean and a configured string). The tool is configured to generate a default constructor and `TOP` static variable storing an instance of the class to use as a return value. We extended OCSEGen to include all public fields and constants and their values in generated complete stubs since these values are used by other application/framework models. Inner classes have to be manually copied into their parent class. The stub was generated in a few seconds.

Stellenbosch University https://scholar.sun.ac.za

3.4 Retaining Side Effects

Modgen can retain behavior of the original class by slicing the byte-code when given a set of fields to retain. It risks generating invalid byte-code that cannot be decompiled/run. OCSEGen can retain side-effects to configured fields using side-effect analysis. The capabilities of the tools are illustrated using two examples from the Android application framework.

The first example for which we applied side-effect analysis is a stub generated for the `MediaPlayer` class shown in Listing 3.2. The `MediaPlayer` class contains many native methods. In order to execute applications referencing the class, an empty-stub is generated for it. But, listeners registered in this class by the application must be retained in models. These listeners are entry-points of the application and must be fired by the driver.

In order to retain them in the `MediaPlayer` class, we configured OCSEGen to retain all side-effects to fields names ending in "listener" and to analyze all methods starting with "set".

When run on the `MediaPlayer` class as the unit, OCSEGen retained the listener fields, their assignment in the set methods and their release in the release method. It could not, however, retain in which method they were fired since this happens in an inner class of the `MediaPlayer`. The final static variables assigned primitive values have also been retained successfully.

Modgen was also used to generate a model for the `MediaPlayer` class. It also retained the listener fields and their assignment, but generated 923 Lines of Code (LOC) with 144 lines that could not be decompiled from byte-code compared to OCSEGen's model of 480 LOC that could compile and run. It retained all fields and lines of code related to the listeners. Additionally, each field to retain had to be specified manually and changes could not be retained across classes.

The next example is the `TextView` class that represents a widget displaying text. All widgets such as the `Button` and `EditText` objects,

Stellenbosch University https://scholar.sun.ac.za


```
1 public class MediaPlayer {
2   private OnCompletionListener mOnCompletionListener;
3   private OnErrorListener mOnErrorListener;
4   private OnPreparedListener mOnPreparedListener;
5
6   public static final java.lang.String MEDIA_MIMETYPE_TEXT_VTT = "
      text/vtt";
7   public static final int MEDIA_ERROR_UNKNOWN = 1;
8   public static final int MEDIA_ERROR_SERVER_DIED = 100;
9   public static final int
      MEDIA_ERROR_NOT_VALID_FOR_PROGRESSIVE_PLAYBACK = 200;
10  public static final int MEDIA_ERROR_IO = -1004;
11
12  public static MediaPlayer TOP = new MediaPlayer();
13
14  public void release(){
15      this.mOnCompletionListener=null;
16      this.mOnErrorListener=null;
17      this.mOnPreparedListener=null;
18      ...
19  }
20
21  public void setOnCompletionListener(OnCompletionListener param0){
22      this.mOnCompletionListener=param0;
23  }
24
25   public void setOnErrorListener(OnErrorListener param0){
26      this.mOnErrorListener=param0;
27  }
28
29   public void setOnPreparedListener(OnPreparedListener param0){
30      this.mOnPreparedListener=param0;
31  }
32
33  static bOnCompletionListener access$800(MediaPlayer param0){
34    return param0.mOnCompletionListener;
35  }
36 ...
37 }
```

Listing 3.2: Extract from the MediaPlayer class

Stellenbosch University https://scholar.sun.ac.za


```
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 3.3: Extract from the TextView Stub created by OCSEGen

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 a calculator application where the text on the buttons are used in the logic of application implementation. Since the `mText` field is an important field in the `TextView` class, OCSEGen was used to identify side-effects to this field. Listing 3.3 presents two methods that side-effect analysis could clearly identify.

This stub is generated using *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 execution paths of the method, this side-effect might not happen.

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, Modgen was also used to slice the `TextView` class given only the `mText` field. Modgen reduced the `TextView` class from 9476 to 5827 LOC but the model that was generated was over-approximated. Its slicing algorithm kept references to all the variables and code affecting this field which resulted in the tool preserving too much of the original code in the model to yield any insight at this

Stellenbosch University https://scholar.sun.ac.za

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 call 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. Java Decompiler (JAD)[1] and Procyon[2] were used but both of these tools are still in development and could not completely decompile the byte-code into compilable Java source-code. Fixing the code manually does not scale to large classes therefore it was not used in the JPF-Android environment model.

3.4.1 Discussion

By investigating these two static analysis tools, we found that both side-effect analysis and slicing can be useful in generating environment models. Side-effect analysis can 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 cannot 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 preserves all functionality in the original class in a sound way by removing unused behavior. But in practice it includes too much of the original behavior of the class to be useful. Another problem with slicing is the capabilities of the slicer and decompilation of the sliced byte-code.

[1] `http://jd.benow.ca`
[2] `https://bitbucket.org/mstrobel/procyon`

Stellenbosch University https://scholar.sun.ac.za


```
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 key = (KeyEvent) bundle.get(Intent.EXTRA_KEY_EVENT);
11
12    switch (key.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 3.4: Extract from RandomMusicPlayer

3.5 Improving Models with Runtime Values

To improve the code coverage achieved by automatically generated models, we introduce an approach to generate drivers and stubs based on values collected during runtime instead of using default values. The approach is motivated by the low code coverage and the number of application crashes when using default stubs for JPF-Android. The following example taken from the RandomMusicPlayer application illustrates the problems with using default stubs. Listing 3.4 is the `MusicReceiver` Broadcast Receiver (BR) implemented to respond to media button presses on the physical device.

BRs are registered for specific Intent objects using *IntentFilters*. These filters are specified in the AndroidManifest.xml file (Listing 3.5) or they can be specified when registering a BR dynamically in the code. IntentFilters specify the action, categories and data that an Intent has to match to be forwarded to the BR. Intents can also contain any number of extra Java

Stellenbosch University https://scholar.sun.ac.za


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

Listing 3.5: Extract from AndroidManifest.xml

```
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 3.6: Extract from Intent.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 3.6).

The MusicReceiver's `onReceive()` method is an entry-point of the application and is called by the JPF-Android driver (Listing 3.4). JPF-Android parses the AndroidManifest file to determine the action and categories of Intents that must be fired. To fire the BR in the above example, the action of the Intent must be set to MEDIA_BUTTON.

JPF-Android has no way of determining what objects should be in the Bundle of the Intent and so null is returned on line 10 of Listing 3.4. The code then crashes on line 12 due to a null-pointer dereferencing exception.

Symbolic execution can be used to analyze methods to determine possible input values, but symbolic execution tools such as Symbolic PathFinder (SPF) have difficulty analyzing this code because of the complex structure of the Intent and Bundle objects.

In Listing 3.7, 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 some

Stellenbosch University https://scholar.sun.ac.za


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

Listing 3.7: Extract from RandomMusicPlayer

other version to ensure that both branches of this if-statement are explored. Our current model of the SystemProperties is an empty-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 3.7 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 analyze the method and stub generation to generate a model using the values it detects.

3.5.1 Collecting and Using Runtime Values

The goal of this approach is to generate events and models using dynamically collected method parameters and return values for testing/analysis. To collect these values, methods in the application (and its libraries) are instrumented 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, logs can be collected over many runs of the application and their inputs/results combined.

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 methods with the same name.

Stellenbosch University https://scholar.sun.ac.za

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 These values are used 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 logs are structured, a regular expression matcher parses the logs into objects stored in an object store or database. These logs can then be queried for the parameters or return values of a specific method. If more than one return value is stored for the same input parameters, they can be returned non-deterministically by the stub. Otherwise, if no values have been stored, a default value is returned. These values are used to improve the driver and stubs to obtain higher coverage of the application code during a follow-up analysis with JPF-Android.

3.5.2 Implementation

Android applications' entry-points are instrumented to collect input parameters for event generation and dependencies' methods are instru-

Stellenbosch University https://scholar.sun.ac.za

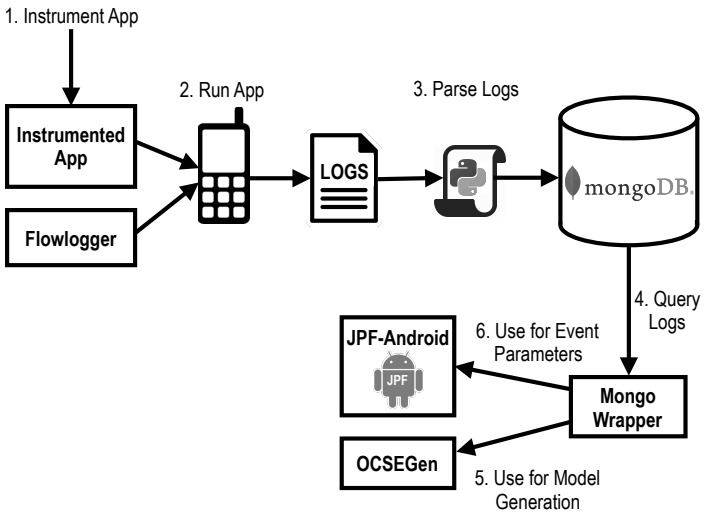


Figure 3.1: Collecting and using runtime values

mented to collect parameters and return values for creating stubs. The main purpose of implementing this approach is to enable the execution of code that could previously not be executed with JPF-Android. The approach is illustrated in Figure 3.1 and each step is described below.

Logging Using Flowlogger We created Flowlogger [10] to simplify logging from the application. Flowlogger contains static methods 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. XStream [21] is used to serialize the input parameters and return values. XStream allows serialization/deserialization of Java objects into XML, JSON or binary data. It can 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 Dalvik Executable (DEX) byte-code during the build process.

Stellenbosch University https://scholar.sun.ac.za

Instrumenting the Byte-Code The most popular approach to instrument Android applications is to use tools such as SOOT [47] and Androguard [3] to instrument the DEX byte-code. But since DEX byte-code differs from Java byte-code, we used the established SOOT static analysis tool 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 value. Lastly, send the method information to Flowlogger to log and return.

Instrumenting native methods is hard since they have no Java implementation. Their method signatures must also be kept intact because it is used to automatically map native method definitions to their C++ implementation. To instrument these methods, a *shadow method* is injected 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.

Collecting and Parsing Logs The instrumented application is run on the emulator and exercised by a user to collect useful values. The log statements generated by Flowlogger are retrieved from the device using Logcat. Logcat is a command-line utility which is part of the Android SDK. The logs collected for different runs and for different applications are parsed and stored in a MongoDB [14] document store. A Python script is used to parse and filter the logs with specified patterns into JSON objects and store them in a MongoDB. The JSON objects should not be confused with the values of their `returnValue` and `params` keys that store Java objects as XML. MongoDB is a highly scalable and fast JSON object store with an extensive RESTFul API.

Stellenbosch University https://scholar.sun.ac.za

Using the Values The logs stored in the MongoDB are retrieved using our custom MongoWrapper library implemented in Java to wrap calls to the DB. XStream, also used by Flowlogger, is then used to deserialize the XML back into Java objects. If there are no recorded results for a method/entry-point, it returns a default value.

We extended OCSEGen to look up runtime values during its code-generation phase to generate more effective stubs. For each method it models, OCSGen looks up a set of unique return objects using the MongoWrapper and XStream. Code is then generated for the method that return these values non-deterministically. In the case of complex objects the tool must be supplied with code to construct an instance of the specific object type given an instance of the object, but primitive values' `toString()` methods are used.

Runtime values collected for entry-points are used in the event generation component of JPF-Android. In this case, events are expanded to a set of events for all observed parameters.

3.5.3 Examples

The first example is from the RandomMusicPlayer Android application shipped with the Android SDK. The RandomMusicPlayer application obtains high coverage results using dynamic testing tools such as monkey and Dynodroid [52], which ensures that we can collect a good set of input parameters and result values.

Listing 3.4 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 3.8). We created a custom converter that excludes Context objects, since the Context is highly dependent on the application and its environment.

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 shown in Listing 3.9. The map of extras in the Intent object contains a `KeyEvent`

Stellenbosch University https://scholar.sun.ac.za


```
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("(Landroid/content/
      Context;Landroid/content/Intent;)V", array);
7
8    // original method body
9
10   FlowLogger.logMethod("com.example.android.musicplayer.
      MusicReceiver", "onReceive", "(Landroid/content/Context;Landroid
      /content/Intent;)V", paramStr, null);
11  }
12 }
```

Listing 3.8: Instrumented MusicReceiver

```
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 3.9: XML generated by XStream for an Intent object

Stellenbosch University https://scholar.sun.ac.za

object with key code 126 which maps to a `KEYCODE_MEDIA_PLAY` event.

The event generator uses the MongoWrapper to look up the set of entry-point parameters given a method signature. The parameters are then converted into Java objects using XStream. When the driver uses of the Intent object generated from the XML in Listing 3.9 as the parameter, the MusicReceiver fires without crashing on line 12 and covers line 16 in Listing 3.4. The more dynamically fired events can be collected, the better coverage can be obtained for this example.

Two examples for which runtime values make a big difference are *SharedPreferences* and *Cursors* since these classes only return primitive values and applications are highly dependent on their return values.

SharedPreferences allow Android applications to store key-value pairs of application settings in XML. The values of these preferences can be changed in a menu or dialog but are most commonly updated using a `PreferenceActivity` displaying all options to the user and allowing them to update their preferences. Instead of allowing the user to change these settings throughout the application at random times, JPF-Android ignores changes to the preferences. Instead it uses a model generated by OCSEGen returning runtime observed preference values non-deterministically. This allows the tool to explore application behavior systematically for all preferences.

An extract from the SharedPreferences stub generated for the GPSLogger application is given in Listing 3.10. In this example each parameter is only mapped to a single return value. Listing 3.11 taken from the SharedPreference stub generated for the Ringdroid application, however, returns different values non-deterministically using the `AndroidVerify.getValues(...)` API of JPF-Android.

Cursors are used by most Android applications to traverse data retrieved from a Content Provider (CP) or database. Neither JPF nor JPF-Android attempts to store the state or backtrack the content in a database. To support the use of cursors, we use a default cursor implementation traversing a data set with a single entry. We then use models generated by OCSEGen mapping specific parameters to runtime return values to im-

Stellenbosch University https://scholar.sun.ac.za


```
1  public java.lang.String getString(java.lang.String param0, java.
       lang.String param1) {
2      String var = param1;
3      if (param0.equals("locale_override")) {
4        var = "";
5      }
6      if (param0.equals("distance_before_logging")) {
7        var = "1";
8      }
9      if (param0.equals("time_before_logging")) {
10       var = "1";
11     }
12     if (param0.equals("new_file_creation")) {
13       var = "onceaday";
14     }
15     if (param0.equals("autoemail_frequency")) {
16       var = "0.08";
17     }
18     if (param0.equals("smtp_server")) {
19       var = "smtp.mail.yahoo.com";
20     }
21     if (param0.equals("smtp_port")) {
22       var = "465";
23     }
24     if (param0.equals("smtp_username")) {
25       var = "testemail";
26     }
27     if (param0.equals("smtp_password")) {
28       var = "1234567";
29     }
30     if (param0.equals("autoemail_target")) {
31       var = "testemail@yahoo.com";
32     }
33     if (param0.equals("osm_accesstoken")) {
34       var = "";
35     }
36     return var;
37   }
```

Listing 3.10: `getString()` method generated with OCSEGen using runtime values

Stellenbosch University https://scholar.sun.ac.za


```
1  public int getInt(java.lang.String param0, int param1) {
2     int var = param1;
3     if (param0.equals("success_count")) {
4       var = (int) AndroidVerify.getValues(new Object[] { 1, 2 },
5           "android.content.SharedPreferences.getInt(
6            success_count)");
7     }
8     if (param0.equals("stats_server_allowed")) {
9       var = 0;
10    }
11    if (param0.equals("stats_server_check")) {
12      var = 2;
13    }
14    if (param0.equals("err_server_allowed")) {
15      var = (int) AndroidVerify.getValues(new Object[] { 1, 2 },
16          "android.content.SharedPreferences.getInt(
17           err_server_allowed)");
18    }
19    if (param0.equals("err_server_check")) {
20      var = 2;
21    }
22    if (param0.equals("stats_server_check")) {
23      var = 2;
24    }
25    return var;
26  }
```

Listing 3.11: `getInt()` method generated with OCSEGen using runtime values

prove the coverage. Listing 3.12 shows a stub generated for the Ringdroid application.

In both of these components the models can be edited manually to throw exceptions or return unobserved values to increase coverage even further.

3.5.4 Discussion and Limitations

This 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.

Stellenbosch University https://scholar.sun.ac.za


```
1 public class CursorRingdroid extends DefaultCursor {
2   public int getColumnIndexOrThrow(java.lang.String param0) {
3     int var = 0;
4     if (param0.equals("title")) {
5       var = 1;
6     }
7     if (param0.equals("artist")) {
8       var = 2;
9     }
10    if (param0.equals("album")) {
11      var = 3;
12    }
13    if (param0.equals("year")) {
14      var = 4;
15    }
16    if (param0.equals("_data")) {
17      var = 5;
18    }
19    return var;
20  }
21
22  public java.lang.String getString(int param0) {
23    String var = "";
24    if (param0 == 0) {
25      var = "1";
26    }
27    if (param0 == 1) {
28      var = "/storage/sdcard/media/audio/ringtones/bensound-cute-
      cut.mp3";
29    }
30    return var;
31  }
32
33  public int getInt(int param0) {
34    int var = 0;
35    if (param0 == 4) {
36      var = (int) AndroidVerify.getValues(new Object[] { 2012, 2010
       }, "android.database.Cursor.getInt(4)");
37    }
38    return var;
39  }
40
41 }
```

Listing 3.12: Cursor stub generated with OCSEGen using runtime values

Stellenbosch University https://scholar.sun.ac.za

The effectiveness of the approach depends on the ability of dynamic analysis tools to reach specific areas in the code where we want to observe runtime values. The main idea is not to focus on getting good coverage with dynamic tools but recording and then optimizing observed values in specific cases. If the dynamic analysis obtains good coverage results for the application in the process, 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 applications.

3.6 Manual Modeling

The Android Open Source Project (AOSP) consists of a large collection of sub-projects. The size, complexity and interdependence of these projects limits the tools that can be used to analyze and understand the platform. We made use of tools such as "grep" and the type and call hierarchy views in Eclipse to navigate the code. Some of the dependencies modeled manually are discussed below.

The Graphical User Interface Each Activity and dialog has a `Window` object storing its GUI as a hierarchy of `View` and `ViewGroup` objects. The hierarchy is usually inflated from XML layout files using the local `LayoutInflator` service in the application framework. Windows are managed by the external `WindowManager` system service. This service stores all windows and tracks the current window to be drawn on the screen. It also maps input events, generated by hardware drivers in response to user events, to the correct `Window` and `View` listeners.

As discussed in Dwyer *et al.* [30] the physical drawing of a GUI on the screen and the visual properties of a GUI do not influence the execution of most applications. For this reason and the fact that JPF-Android uses a custom event generator, the `WindowManagerService`, `Window` and `View` classes can be greatly simplified. Therefore, a local `WindowManager` is implemented manually to store the history of windows for when an

Stellenbosch University https://scholar.sun.ac.za

Activity finishes and it needs to move back to the previous `Window`. JPF-Android also includes a custom layout inflater inflating views using only properties required to execute the application correctly, namely: enabledness, visibility, input values and listeners. GUI component stubs are generated using OCSEGen [61] and extended manually to collect and fire the entry-points of the component.

Certain widgets require inputs entered by a user during execution. To reduce the number of possibilities of these inputs and to reduce the number of events fired by the driver, widgets return predefined configurable values by default. Inputs are either selected randomly or non-deterministically from a list depending on JPF-Android's configuration.

Component Life-cycle Management And Interaction Android application components can interact with each other or with external components. The `ActivityManagerService` is responsible for storing all running application components and facilitating interactions between components across and within applications. Since we only verify a single application, this service is modeled as a local service that keeps track of all the running components of the SUT to enable component interaction and to detect events for running components during event generation. Requests to other applications are modeled by returning configured `Intents`.

Resource Management The `Resources` class is also modeled manually to parse the actual resource values (Strings, int, float, layout files, images) stored in XML in order for the application to function correctly.

File Management JPF-Android is run in a Docker [6, 11] container which allows us to set up the file system structure the same as on a device. We then use JPF's file models to access and create files. We should note, however, that files that should exist must be added to the file system manually or in some cases we use custom file models to return specific results such as `FileNotFoundException` or `IOException`.

Stellenbosch University https://scholar.sun.ac.za

3.7 The Android Environment Model

The original Android application framework, excluding native and operating system services, exposes 1402 classes. Therefore, we approached building an environment model for Android applications as an iterative process that will continuously improve as models are reused and extended to support new behavior required by applications. We started with 50 models in order to run a basic Android application. Over the course of the last few years we created 417 models totaling around 32kLOC. JPF-Android is open-source and the models are available on Bitbucket [3]. These models can be adapted to be used for JUnit testing.

Table 3.1 lists other commonly used dependencies in the Android environment and how they are modeled.

3.8 Summary

In this chapter we discussed the challenges to environment modeling for Android applications and presented the different approaches used to create an environment model for Android applications. In the next chapter, we discuss how to drive the execution of Android applications using event generation.

[3] Available at: `https://bitbucket.org/heila/jpf-android/src` and `https://bitbucket.org/heila/jpf-android-examples/src`

Table 3.1: Common dependency models

Service	Description	Model
ResourceManager	Retrieve Strings, images or raw data	Retrieves values from XML or uses default values
SystemProperties	Lookup underlying system properties	Default models using runtime values
LayoutInflator	Inflates GUI from XML layout files	Reusable manual model inflating GUI from XML
MediaPlayer	Play media files	Reusable manual model from default models
ContentRetriever	Browse data associated with URI using cursor	Reusable manual model mapping URI to cursor — default model for cursor using runtime values
DatabaseInterface	Access and query a DB	Reusable manual model based on models to retain side effects — returns cursor utilizing runtime values
HTTP Requests	Make complex HTTP requests	Default models returning runtime values
File I/O	Access files on storage	Reuse JPF models, mount `/storage` to resolve paths in Docker

Stellenbosch University https://scholar.sun.ac.za

Stellenbosch University https://scholar.sun.ac.za

Chapter 4

Event Generation

Android applications have an event-driven design. Applications register *entry-points* in their environment triggered by *events* to drive the execution of the application. These events can be combined into millions of different event sequences of potentially unbounded lengths. Each of these sequences may trigger erroneous application behavior. In this chapter we discuss how entry-points are detected and events generated and fired by JPF-Android as well as the different strategies employed by JPF-Android to generate event sequences.

4.1 Generating and Processing Events

Android applications contain entry-points in the form of listeners and callbacks registered in local/external services in the application framework. These entry-points include Graphical User Interface (GUI) listeners, component life-cycle callbacks as well as more specific callbacks registered for location updates, incoming calls, phone state changes or sensor updates. Applications are executed by firing their entry-point methods with specific parameters. Local and external services, in which the entry-points are registered, are modeled in JPF-Android. In order to generate and fire events to execute these entry-points, JPF-Android requires the services to store registered application callbacks and listeners and provide methods

Stellenbosch University https://scholar.sun.ac.za

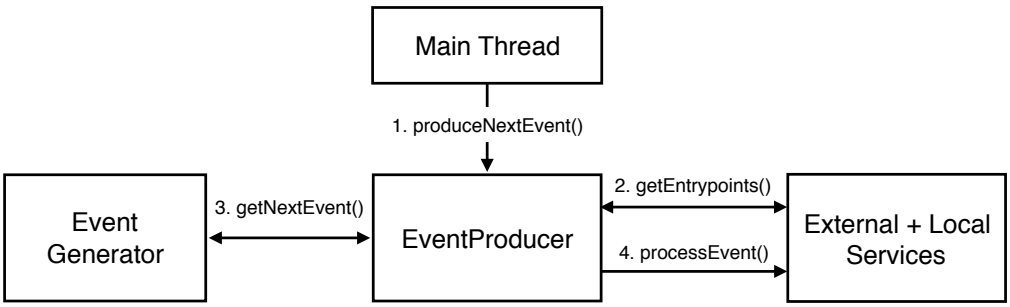


Figure 4.1: Event generation architecture

in order to retrieve and process events. The services are then registered to receive specific event types.

Events generated by JPF-Android consist of a *type* to look up the services that can process the event, a set of *parameters* to identify the entry-point and its arguments and the *window name* for verification purposes. JPF-Android supports the following event types:

UIEvent Fires View listeners: onClick, onItemSelected, onLongClick, etc. for GUI components

SystemEvent Events that start/send Intents to application components

KeyPressEvent Fires onKeyDown and onKeyUp listeners in Activities/GUI

MenuItemEvent Fires menu item selected callbacks in Activity

SharedPreferencesEvent Fires preference change listeners

LocationEvent Fires callbacks registered for location updates

In the Android application framework the main event processing thread of the application (the `Looper`) retrieves events from its message queue. The message queue contains callbacks added by other threads or asynchronous method calls of the application itself.

In order to produce events JPF-Android extends the `Looper` to also request events from the `EventProducer` (Step 1 in Figure 4.1). Events can be produced at any time, but to reduce scheduling possibilities, the

Stellenbosch University https://scholar.sun.ac.za

`EventProducer` is only called when the message queue is empty, and other application threads are waiting. At this point, the application usually waits for events from the Android system or from the user to continue executing, but instead it now processes the next event from the `EventProducer`. The `EventProducer` is started together with the other system services.

The `EventProducer` produces a new event by looping through all registered services and collecting a list of enabled entry-points and their default parameters (Step 2). These entry-points are then passed to the configured event generator to generate a set of next events (Step 3). The set of returned events are fired non-deterministically by the `EventProducer`. This functionality branches the execution for each of the possible events and visit each of them in turn using state matching and backtracking. The current event being processed is forwarded to all services registered for the specific event type (Step 4).

When no more events can be generated, and all application threads have completed, the application's state is expected to match and the search ended in this path.

4.2 Event Generation Strategies

JPF-Android supports custom event generators enabling it to use different event generation strategies. By default the tool implements the following event generation strategies: default, heuristic and scripting.The next sections describe the three different event generators.

4.2.1 Default Event Generator

The default event generator generates event sequences dynamically, at runtime. JPF-Android can reduce the number and length of the discovered sequences using state matching to stop execution if a previous state that has already been explored is reached again. The default event generator explores all possible events with default parameters for a given set of

Stellenbosch University https://scholar.sun.ac.za

entry-points. It provides a simple way to explore/discover events and hard-to-reach code of an unknown application.

4.2.2 Heuristic Event Generator

In theory we want to explore all possible event sequences, but in practice — even with state matching — this is not always possible in an acceptable time or with the given resource limitations. The heuristic event generation strategy extends the default strategy but limits the number of times an event can occur per path. By default it filters events by allowing only a single instance of an event to be fired per branch. Events with different parameters are seen as different events. This generator reduces the length and number of explored event sequences.

4.2.3 Script Event Generator

The script event generator allows users to write scripts containing sequences of events [62]. Scripting event sequences is useful to analyze specific application behavior that might be hard to reach or may require specific environment configurations. This approach limits the environment and application behavior to allow a more exhaustive exploration of the application. Writing scripts, however, requires in-depth knowledge of the application and its environment.

JPF-Android's scripts are based on the scripting language introduced by JPF-AWT. The scripting language is a dynamically typed, interpreted language. JPF-Android introduces two new constructs: `Sections` and `Groups` and expands the functionality of the `Any` and `Repeat` constructs. The syntax of this language is described in the next section. JPF-Android provides its own parser and interpreter for the language as part of the script event generator to support this new behavior.

Stellenbosch University https://scholar.sun.ac.za

4.2.3.1 Model Checking Considerations

The previous two event generators generate events on-the-fly depending on the state of the System-Under-Test (SUT). When using a script we need to keep a consistent state when backtracking. Therefore, the current position in the script is included in the state. The current iteration of repeat elements and the current choice selected in an `Any` element are also included in the state. The state can only match when events scheduled non-deterministically lead to the same state after the `Repeat`/`Any` script element. In the case of an infinite `Repeat` the state can match after two or more repetitions since JPF-Android does not keep a counter of the number of loop iterations as part of the state.

4.2.4 Configurable Properties of Event Generators

The event generators can be configured in the follow ways:

Filtering Events can be excluded/included using lists of regular expressions. This allows filtering certain events.

Depth Bound The maximum length of generated event sequences can be specified.

Dynamic Event Parameters Using correct parameters for events is important since incorrect parameters can lead to false positives (crashes that should not occur in the entry-point code). Service models define default parameters of entry-points, but these values can be improved by enabling dynamic event generation where generated events are refined using pre-collected runtime (as discussed in 3.5), manually or statically collected values stored in a database. The user can configure the event generator to expand each event to a set of events returning different runtime collected parameters. In this case the database is queried for the entry-point method of the event and a new event is generated for each unique set of parameters observed. This option quickly explodes the state space and should

Stellenbosch University https://scholar.sun.ac.za

not be used for all entry-points but only to improve coverage for selected entry-points.

4.3 The Input Script

The grammar for the scripting language used by input scripts is given in Figure 4.2. JPF-Android divides the input script into `Sections`. Each `Section` in the script groups together the input events of a specific Window. The reason for this is that Android applications contain many Windows, at least one for each Activity. A script, which contains non-deterministic input events, can non-deterministically switch between Windows. If the script continues to execute sequentially after a possible Window switch, the next events might not necessarily be relevant to the current Window on the screen. The JPF-AWT project only verifies a single Window so this problem was not considered at that time.

A `Section` defines a unique name identifying the Window to which its input events are bound. The main Window of an Activity needs to be identified by the name of the Activity. If multiple Activities in the application have the same name, the name needs to be prefixed by the package name of the Activity. The name of the initial `Section` where the script starts executing is set to "default".

The `Repeat` construct simplifies the script by allowing a user to script a sequence of events that is repeated automatically a certain number of times. Listing 4.1 shows an example of the `Repeat` construct together with the event sequence generated by the scripting environment in Listing 4.2.

```
1 REPEAT 2 {
2   eventA()
3   eventB()
4 }
```

Listing 4.1: Repeat example

```
eventA()
eventB()
eventA()
eventB()
```

Listing 4.2: Generated events

Stellenbosch University https://scholar.sun.ac.za

⟨*script*⟩ = 'SCRIPT' { ⟨*section*⟩ }.

⟨*section*⟩ = 'SECTION' ('default' | ⟨*id*⟩) '{' { ⟨*sequence*⟩ } '}'.

⟨*sequence*⟩ = ⟨*iteration*⟩ | ⟨*selection*⟩ | ⟨*event*⟩.

⟨*iteration*⟩ = 'REPEAT' ⟨*num*⟩ '{' ⟨*sequence*⟩ '}'.

⟨*selection*⟩ = 'ANY' '{' ⟨*group*⟩ { ',' ⟨*group*⟩ } '}'.

⟨*group*⟩ = { ⟨*sequence*⟩ }.

⟨*event*⟩ = ⟨*uievent*⟩ | ⟨*sysevent*⟩

⟨*uievent*⟩ = '$' ⟨*target*⟩ '.' ⟨*action*⟩ '(' ⟨*params*⟩ ')'.

⟨*sysevent*⟩ = ['@' ⟨*target*⟩ '.' | 'device' '.'] ⟨*action*⟩ '(' ⟨*params*⟩ ')'.

⟨*target*⟩ = ⟨*id*⟩.

⟨*action*⟩ = ⟨*id*⟩.

⟨*params*⟩ = [⟨*param*⟩ {, ⟨*param*⟩ }].

⟨*param*⟩ = '@' ⟨*target*⟩ | ⟨*target*⟩ | ⟨*string*⟩ | ⟨*float*⟩.

⟨*id*⟩ = ⟨*letter*⟩ { ⟨*letter*⟩ | ⟨*digit*⟩ | '_' }.

⟨*num*⟩ = '*' | ⟨*digit*⟩ { ⟨*digit*⟩ }.

⟨*string*⟩ = '"' ⟨*char*⟩ { ⟨*char*⟩ } '"'.

⟨*float*⟩ = ⟨*digit*⟩ { '.' | ⟨*digit*⟩ }.

Figure 4.2: EBNF of the scripting language

JPF-Android extends the `Repeat` construct to allow infinite repeats by specifying a "*" as the number of repetitions. In this case the execution of the application is only stopped if a state match occurs in each branch of the application or the maximum search depth is reached. `Repeat` constructs can now contain `Any` script elements.

In JPF-AWT the `Any` construct contains a list of events scheduled to execute non-deterministically. JPF-Android extends the `Any` construct to contain a list of `Group` objects, each containing a list of events. Instead of only scheduling the events non-deterministically, JPF-Android now schedules the `Groups` to execute non-deterministically. JPF-Android also

Stellenbosch University https://scholar.sun.ac.za

extends these `Groups` to contain cascaded `Any` and `Repeat` constructs.

Figure 4.3 shows an example of using the `Any` construct together with the event sequence generated by the scripting environment.

```
1 eventZ()
2
3 ANY {
4     GROUP {
5         eventA()
6         eventB()
7     }
8     GROUP {
9         eventC()
10    }
11 }
12
13 eventD()
```

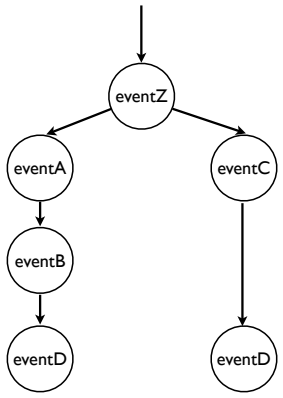


Figure 4.3: Example of an `Any` script element

The script supports two types of events: user events and system events.

4.3.1 User Events

User or User Interface (UI) events are triggered by a user interacting with GUI elements on the screen or with the physical buttons on the device. The input script allows the user to simulate UI events with specific parameters. The syntax for scripting UI events is given in Figure 4.2 and consists of three main parts:

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

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

params optional, comma separated list of parameters

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

Stellenbosch University https://scholar.sun.ac.za


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

Listing 4.3: Examples of changing the system state

```
1 @WifiOffIntent.setAction("android.net.conn.CONNECTION_CHANGE")
2 @WifiOffIntent.putExtraString("type", "WIFI")
3 @WifiOffIntent.putExtraString("state", "DISCONNECTED")
4 @WifiOffIntent.putExtraString("reason", "NO SIGNAL")
```

Listing 4.4: Creating a @WifiOffIntent in the script

4.3.2 System Events

Android applications are also driven by system events. System events are fired by the Android system in response to a change in the system state or by other applications interacting with this application. They include notifications such as the state of the WiFi connection changing when the WiFi signal drops. The syntax of system events is given in Figure 4.2.

The script event generator enables the user to change the state of the system to induces system events. Examples of these state change events are given in Listing 4.3.

State changes can also be induced from the script by constructing custom Intents. Intent objects are identified in the script by starting with an "@" symbol. The properties of the Intent object are set by using the EBNF definition of an event to call the setter methods of the Intent. In Listing 4.4 a custom Intent, `@WifiOffIntent` is constructed to disable the WiFi radio.

Intents constructed in the script have to contain all the necessary information the application expects from the specific type of Intent. The Intent is then broadcast to the Android system using the sendBroadcast system event in the script. Listing 4.5 shows how the Intent is sent to the system.

Stellenbosch University https://scholar.sun.ac.za


```
1 sendBroadcast(@WifiOffIntent)
```

Listing 4.5: Sending a WifiOffIntent Intent from the script

The script event generator also predefines common Intents that are used frequently to simplify the script. These Intent objects include Intents used to start an Activity, disable/enable the WiFi and setting the battery status. Custom Intents allow the user to provide more details on the action/event.

4.4 The Calculator Example

One of the Android applications verified on JPF-Android is a scientific calculator. The calculator has two Activities: a simple Activity that displays basic arithmetic operations and a scientific Activity that displays more complex arithmetic operations (Figure 4.4.) We use this example to illustrate the functionality of the script event generator.

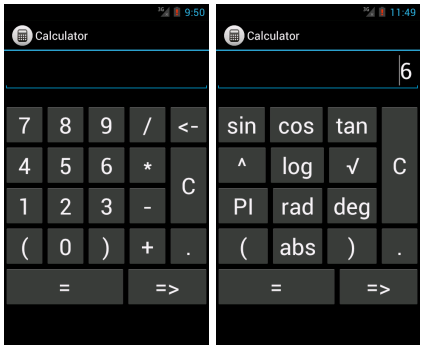


Figure 4.4: The two Activities of the Calculator application. The SimpleActivity is shown on the left and the ScientificActivity is on the right.

The user can switch between these Activities using the "=>" button on the interface. When the user switches Activities, the current expression the user entered into the input box, has to be sent to the next Activity. This

Stellenbosch University https://scholar.sun.ac.za

information is sent using an `Intent` object transfered between Activities when starting the next Activity.

The calculator makes use of an expression evaluator[1] to evaluate the mathematical expressions entered by users when the equals button is pressed. When the user divides a value by zero, an `ArithmeticException` is thrown by the expressions evaluator.

Two runtime errors are injected into the application code. The first error is injected into the code by removing the try-catch block for this runtime exception. The Arithmetic exception then causes the application to crash (See Figure 4.5).

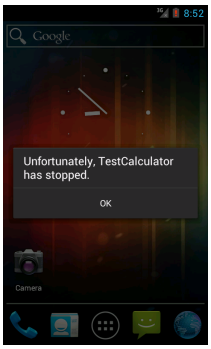


Figure 4.5: The calculator application crashing

The second error injected into the application is neglecting to attach the state information to the `Intent` object passed to the next Activity. When the user switches to the other Activity, a `NullPointerException` is thrown when the application tries to read the state information from the `Intent`.

Both of these errors are difficult to detect with unit testing. The `ArithmeticException` is challenging due to the many possible input sequence combinations. If a test case did not specifically identify this as a point of interest, unit testing would not have detected this error. The second error is challenging since it only occurs when the interaction between Activities is tested.

[1] Available at `https://github.com/uklimaschewski/EvalEx`

Stellenbosch University https://scholar.sun.ac.za


```
1 #--- dependencies on other JPF projects
2 @using = jpf-android
3
4 #--- target setup
5 target = com.example.calculator.SimpleActivity
6 classpath+=${jpf-android}/../Examples/Calculator/bin/classes/;
7 sourcepath=${jpf-android}/../Examples/Calculator/src;
8
9 #--- android setup
10 android.script=${config_path}/Test1.es
11
12 #--- listeners
13 vm.halt_on_throw=java.lang.ArithmeticException
```

Listing 4.6: Properties file of the Calculator

```
1 SECTION default {
2    @startIntent.setComponent("com.example.calculator.SimpleActivity
      ")
3    startActivity(@startIntent)
4 }
5
6 SECTION com.example.calculator.SimpleActivity {
7    $button[0-9].onClick()
8    $button<Plus|Minus|Mul|Div>.onClick()
9    $button[0-9].onClick()
10   $buttonEquals.onClick()
11 }
```

Listing 4.7: Input script for Test 1 of the Calculator application

4.4.0.1 Test 1: Detecting the Arithmetic Exception

The JPF properties file for detecting the `ArithmeticException` is given in Listing 4.6. The target property is set to the launcher Activity of the application, the SimpleActivity. Scripts are stored in files with ".es" extensions in the project directory. The name of the currently in-use script is configured in the Java PathFinder (JPF) property file of the SUT as the "android.script" property. The input script is set to `Test1.es` on Line 10. On Line 13, JPF is set to stop execution if an arithmetic exception is thrown by the application.

The script is shown in Listing 4.7. It uses a shorthand syntax on Line 7-9 to more compactly create `Any` constructs across a range of buttons

Stellenbosch University https://scholar.sun.ac.za

($\$button[0-9]$ and $\$button < Plus|Minus|Mul|Div >$). The execution starts in the default section of the script. In the default section the component of the `startIntent` is set to "SimpleActivity" indicating that this is where JPF must start the execution. This Intent is then sent to the ActivityManager of the system by using the `startActivity` system event. After the SimpleActivity has been started by JPF-Android, the application requests the next input event in the message queue. At this point the message queue is empty and an event is requested from the script event generator. Since the current Activity is the SimpleActivity, the next event in the SimpleActivity `Section` of the script is returned. This event is a compacted version of the `Any` construct: `$button[0-9].onClick()`. This event is parsed into an `Any` construct containing ten `Groups`. Each `Group` contains an onClick event for one of the buttons 0-9. As none of the events in the SimpleActivity section switches between Windows of the application, the input events in the SimpleActivity section of the script is executed one-by-one until the last event `$buttonEquals.onClick()` is processed. After this event, JPF's state matches since no more events are available.

When the application is run on JPF-Android using this script, it detects the `ArithmeticException` due to the division by zero. The results printed out by JPF-Android is given in Figure 4.6. Firstly, JPF-Android gives the execution trace to where the error occurred in the application code. It then also gives the list of input events that lead to the error.

4.4.0.2 Test 2: Detecting the NullPointerException

Next, a `NullPointerException` was introduced into the application. The property file for this example is similar to the one shown in Listing 4.7 except that the `vm.halt_on_throw` property is set to detect null pointer exceptions (see Listing 4.8).

Since this test verifies the transition between the Activities, the script contains an additional section for the ScientificActivity. The script for this example is shown in Listing 4.9.

Stellenbosch University https://scholar.sun.ac.za


```
====================================================== error 1
gov.nasa.jpf.vm.NoUncaughtExceptionsProperty
java.lang.ArithmeticException: Division undefined
at java.math.BigDecimal.divide(BigDecimal.java:1668)
at com.udojava.evalex.Expression$4.eval(Expression.java:490)
at com.udojava.evalex.Expression.eval(Expression.java:849)
at com.example.calculator.CalculatorActivity.calculate(CalculatorActivity.java:137)
at com.example.calculator.CalculatorActivity.onClick(CalculatorActivity.java:98)
at android.view.View.onClick(View.java:463)
at java.lang.reflect.Method.invoke(gov.nasa.jpf.vm.JPF_java_lang_reflect_Method)
at android.view.Window.handleViewAction(Window.java:208)
at android.view.WindowManager.handleViewAction(WindowManager.java:50)
at android.os.MessageQueue.getNextScriptAction(...)
at android.os.MessageQueue.next(MessageQueue.java:59)
at android.os.Looper.loop(Looper.java:88)
at android.app.ActivityThread.main(ActivityThread.java:2197)
at android.os.ServiceManager.start(ServiceManager.java:73)
at com.example.calculator.SimpleActivity.main(SimpleActivity.java:0)
====================================================== error input sequence

@startIntent.setComponent("com.example.calculator.SimpleActivity")
startActivity("@startIntent")
$button0.onClick()
$buttonDiv.onClick()
$button0.onClick()
$buttonEquals.onClick()

====================================================== results
error #1: gov.nasa.jpf.vm.NoUncaughtExceptionsProperty
    "java.lang.ArithmeticException: Division undefined ..."

====================================================== statistics
elapsed time:       00:00:12
states:             new=1287, visited=0, backtracked=1286, end=30
search:             maxDepth=12, constraints hit=0
choice generators:  thread=837 (signal=0, lock=1, shared ref=0), data=51
heap:               new=310959, released=42303, max live=2931, gc-cycles=1286
instructions:       12654542
max memory:         300MB
loaded code:        classes=231, methods=4140

====================================================== search finished
```

Figure 4.6: Test 1 results

When this example is run, the SimpleActivity is started and then a "1 – " is entered into the input box. The `nextButton` is then pressed to switch to the ScientificActivity. At this point the SimpleActivity normally bundles the expression in the input box, "1 – ", with the Intent to start the ScientificActivity. When the ScientificActivity is started, it tries to retrieve this information from the Intent. An uncaught `NullPointerException` is then thrown as this information is not attached to the Intent. JPF-

Stellenbosch University https://scholar.sun.ac.za


```
1 #--- listeners
2 vm.halt_on_throw=java.lang.NullPointerException
```

Listing 4.8: Property file of Test 2

```
1 SECTION default {
2   @startIntent.setComponent("com.example.calculator.SimpleActivity"
      )
3   startActivity(@startIntent)
4 }
5
6 SECTION com.example.calculator.SimpleActivity {
7   $button1.onClick()
8   $buttonMinus.onClick()
9   $buttonNext.onClick()
10 }
11
12 SECTION com.example.calculator.ScientificActivity  {
13   $button<Sin|Cos|Tan>.onClick()
14   $buttonOpenParenthesis.onClick()
15   $buttonPI.onClick()
16   $buttonCloseParenthesis.onClick()
17   $buttonEquals.onClick()
18 }
```

Listing 4.9: Script for Test 2

Android's results are shown in Figure 4.7.

4.4.1 The Deadlock Example

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

Stellenbosch University https://scholar.sun.ac.za


```
====================================================== error 1
gov.nasa.jpf.vm.NoUncaughtExceptionsProperty
java.lang.NullPointerException:
Calling 'getString(Ljava/lang/String;)Ljava/lang/String;' on null object
at com.example.calculator.CalculatorActivity.restoreState(CalculatorActivity.java:70)
at com.example.calculator.ScientificActivity.onCreate(ScientificActivity.java:17)
at android.app.Activity.performCreate(Activity.java:1838)
at android.app.Instrumentation.callActivityOnCreate(Instrumentation.java:361)
at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:1186)
at android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:1243)
at android.app.ActivityThread.access$400(ActivityThread.java:97)
at android.app.ActivityThread$H.handleMessage(ActivityThread.java:642)
at android.os.Handler.dispatchMessage(Handler.java:53)
at android.os.Looper.loop(Looper.java:103)
at android.app.ActivityThread.main(ActivityThread.java:2197)
at android.os.ServiceManager.start(ServiceManager.java:73)
at com.example.calculator.SimpleActivity.main(SimpleActivity.java:0)

====================================================== error input sequence

@startIntent.setComponent("com.example.calculator.SimpleActivity")
startActivity("@startIntent")
$button1.onClick()
$buttonMinus.onClick()
$buttonNext.onClick()

========================================================== results
error #1: gov.nasa.jpf.vm.NoUncaughtExceptionsProperty
    "java.lang.NullPointerException: Calling 'getString..."

========================================================== statistics
elapsed time:       00:00:01
states:             new=10, visited=0, backtracked=0, end=0
search:             maxDepth=10, constraints hit=0
choice generators:  thread=10 (signal=0, lock=1, shared ref=0), data=0
heap:               new=4743, released=2288, max live=2438, gc-cycles=9
instructions:       95826
max memory:         61MB
loaded code:        classes=189, methods=3723

========================================================== search finished
```

Figure 4.7: Test 2 results

given in Listing 4.10. JPF is configured to listen for deadlocks and schedules the threads in all possible ways to detect the deadlock. The results returned by JPF-Android are given in Figure 4.8.

Stellenbosch University https://scholar.sun.ac.za


```
1 SECTION default {
2   @startIntent.setComponent("com.example.jpf.DeadlockActivity")
3   startActivity(@startIntent)
4 }
5
6 SECTION DeadlockActivity  {
7     $button1.onClick()
8     $button2.onClick()
9 }
```

Listing 4.10: Script for deadlock application

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

Figure 4.8: Results of the Deadlock Activity

4.5 Summary

This section discussed how JPF-Android generates and processes events to drive the execution of Android applications. The next section looks at how to apply model checking to Android applications effectively.

Stellenbosch University https://scholar.sun.ac.za

Chapter 5

Model Checking

Android applications have a multi-threaded, event-driven, asynchronous message-passing design which causes them to have too many paths to explore. Explicit-state model checking with Java PathFinder (JPF) allows more efficient exploration of Android applications. JPF supports *non-deterministic exploration* of choices and implements *state matching and backtracking* to bound and reduce the search space. It also exposes a property and listener Application Programming Interface (API) to track the execution of the application at the byte-code level. In this chapter we describe how JPF is optimized and extended in JPF-Android to improve the coverage and reduce the analysis size for Android applications.

5.1 Non-deterministic Choices

Android applications have many environment dependencies and respond to many different event parameters that determine which path the application's execution follows. JPF can explore multiple paths of an application non-deterministically using state storage and backtracking. When the execution reaches a point with multiple paths forward, the set of possible choices are recorded in a *ChoiceGenerator* and the state of the application is stored. The different paths are then explored one-by-one, backtracking the application state to the stored version for each choice.

Stellenbosch University https://scholar.sun.ac.za

By default JPF uses choices to explore multiple thread interleavings. For JPF-Android, however, we utilized this behavior to explore three types of choices: *thread choices*, *event choices* and *environment data choices*.

5.1.1 Thread Choices

Thread choices are used to explore different thread interleavings to find deadlocks and race conditions. These choices are managed by JPF and used to detect race conditions and deadlocks. JPF explores all thread interleavings to ensure the application does not reach a state where threads are waiting, blocked or cause a race condition. Exploring all possible thread paths is a resource intensive process.

JPF-Android includes custom thread synchronization policies that only explore a single thread interleaving. This reduces thread choices when not searching for concurrency errors. JPF-Android provides the following policies:

Random Randomly chooses a single thread schedule,

Priority Chooses the thread with the highest priority, started last. The main event handling thread is given a low priority to ensure other threads complete or wait before continuing with it.

Although random scheduling can achieve better coverage, the application must be run multiple times to achieve a consistent coverage percentage. For this reason the tool uses Priority scheduling by default returning the last started thread until it is done or waiting after which it continues to the previous thread.

5.1.2 Event Choices

When the main message queue is empty and the application is idle, the main thread collects a set of next events from the event producer. These events are scheduled non-deterministically introducing an event choice point in the execution. Each of these events is explored in its own path.

Stellenbosch University https://scholar.sun.ac.za

5.1.3 Environment Data Choices

Environment data choices enable systematic exploration of different return values in environment models. Choices are explored non-deterministically instead of selecting one randomly. This enables obtaining higher coverage with shorter event sequences. The number of states, however, increases with each additional choice point reached in the path. To improve the scalability of the analysis for large applications with many choices, JPF-Android can be configured to explore each unique choice point only the first time it is reached in a path whereafter the value is cached for the rest of the path. When the choice point is reached again, the cached value is returned and the path is not branched. All choice combinations are not explored in this case, but the analysis scales much better and JPF-Android analyzes all choices at least once.

Listing 5.1 shows how environment data choice points can be created using JPF-Android's `AndroidVerify` API. The `AndroidVerify` API contains a set of methods to create labeled choice points that schedule a set of choices non-deterministically. The `getBoolean` method on line 5 in Listing 5.1, for example, non-deterministically returns true and false. The parameter given to the API method is the ID that uniquely identifies the choice point and is used to cache and retrieve its value.

5.2 State Matching

State matching bounds the execution of a System-Under-Test (SUT) by stopping exploration of paths when reaching a previously visited state. Software model checkers, such as JPF, apply model checking to programs written in modern languages with a large number of potentially complex states. Android applications contain many different choices which lead to an increased number of states. Therefore, JPF-Android adds optimizations to reduce the state space explored by JPF. We identified two main areas that cause unnecessary state explosion: *class loading and serialization* and *unbounded variables*. In the next two sections the problems and proposed

Stellenbosch University https://scholar.sun.ac.za


```
1  @Override
2  public boolean getBoolean(String key, boolean defValue) {
3
4    if (key.equals("sort_key")) {
5      return AndroidVerify.getBoolean("Pref(" + "sort_key" + ")");
6
7    } else if (key.equals("omitbackup")) {
8      return AndroidVerify.getBoolean("Pref(" + "omitbackup" + ")");
9
10   } else if (key.equals("keyfile")) {
11     return AndroidVerify.getBoolean("Pref(" + "keyfile" + ")");
12
13   } else if (key.equals("maskpass")) {
14     return AndroidVerify.getBoolean("Pref(" + "maskpass" + ")");
15   }
16
17   return defValue;
18 }
19
20 public URL getNextUrl() throws MalformedURLException {
21
22   boolean b = AndroidVerify.getBoolean(
23               "FacebookLogin.getFullLoginUrl()?");
24   if (b) {
25     return new URL("http://www.facebook.com/cancel");
26   } else {
27     return new URL("http://www.facebook.com/login");
28   }
29
30 }
```

Listing 5.1: AndroidVerify API example

solutions to optimize state matching are discussed.

5.2.1 Class Loading and Serialization

JPF-Android breaks the current transition before each event and environment data choice to trigger checking for a state match. This enables it to stop exploration when a choice has no side-effects on the state of the system or matches a previously visited state. The application state used for state matching includes a static area storing the list of loaded classes and the values of their static variables. When a new class is loaded, the length of this list increases, which in turn changes the state of the SUT — even if the class has no static fields.

The disadvantage of storing the list of loaded classes in the state is

Stellenbosch University https://scholar.sun.ac.za

that the state can change just because a new class was loaded — even if the class has no static fields. This results in JPF-Android exploring events twice before state matching.

An example of such a case is a button that shows a toast in an Android application (Listing 5.2.) A user cannot interact with a toast. It is a message displayed for a few seconds on the screen and does not change the state. Before and after the button is pressed the state of the application is the same, but since the `Toast` class was loaded, the state changed so the button needs to be pressed again for a state match to occur. This results in an explosion in the number and length of event sequences.

```
1 Button b = ...;
2 b.setOnClickListener(new OnClickListener() {
3   public onClick() {
4     if (!networkConnected()) {
5       Toast.makeText(getActivity(), "Error updating: no network
      connection.", Toast.LENGTH_LONG).show();
6       return;
7     }
8     // update
9   }
10 }
```

Listing 5.2: Toast example

Ideally, the state should not change when classes are loaded, but only when their static fields are changed from their default values set at class initialization. To overcome this problem, JPF-Android preloads classes before the application executes. In this case the default values of static fields are already set and will only influence the state when updated. Preloading classes that contain choices in their static initialization code, however, causes the entire application to be executed multiple times for each choice. We minimize choices in this area by using fields instead of static variables in the model environment.

Preloading classes causes the number of classes serialized for state comparison to sharply increase. We noticed that only a small subset of loaded classes in the Android environment contain non-final static

Stellenbosch University https://scholar.sun.ac.za

fields (or any static fields) that can be updated. Therefore, to reduce the number of classes in the state, we created the `OptimizedStateSerializer`. This serializer marks classes that should be serialized at load time. Classes with no static fields or only static final fields with constant values (that cannot change) are then not serialized as part of the state. The `OptimizedStateSerializer` also filters out final static fields for classes included in the state.

5.2.2 Unbounded Variables

Software applications are not inherently finite state by design which leads to non-termination during model checking. To model check programs, the application and its environment must be abstracted to form a finite-state system. Counters and variables keeping track of history, or that continuously change, prevents state matching. Their values should be bound or excluded from the state to form a finite state system. Often in-depth knowledge of a software system is required to identify and abstract variables causing a state-space explosion. Abstracting an SUT to form a finite-state system has been studied before, but software tools developed for this purpose typically require the user to manually identify fields to be abstracted [38, 61].

Listing 5.3 illustrates this problem using the driver of a binary tree that contains an implicit counter. The driver adds an integer to the tree and then removes the integer again and asserts that the tree is empty. This code is placed in a while-loop and the transition is ended and the state stored after each iteration for state matching. When running this application on JPF one expects the application to match states after the second iteration of the loop. At this point the binary tree contains no elements — the same state it was in after the first iteration.

This example, however, does not terminate. This indicates that the state changed for each iteration of the loop. Using manual state comparison one can see that the reason for this is the unbounded `modCount` field in the `BTree` class. This field is a modification counter used to catch

Stellenbosch University https://scholar.sun.ac.za


```
1 public static void main(String[] args) {
2   BTree<Integer> bt = new BTree<Integer>();
3   while (true) {
4     bt.push(5);
5     bt.remove(5);
6     assert bt.size == 0;
7     Verify.breakTransition();
8   }
9 }
```

Listing 5.3: Binary tree driver

modifications to the tree while iterating over its nodes using an Iterator. The modification counter is increased each time an integer is added or removed from the tree. If `modCount` is removed from the state using JPF's `@FilterField` annotation, the system executes as expected and the state matches after the second iteration of the loop. `modCount` is used so often in data structures in Java (Lists or Maps) that JPF automatically filters the field from the state for the `java.util` package.

Counters may be simple to identify manually in small examples but the Android framework and the Android environment model contain many of these variables causing a state explosion and leading to the analysis never completing in JPF-Android. We implemented *State Comparator* to identify variables that cause the SUT to have too many states for the model checker to explore. Removing these variables from the state, or abstracting/bounding their values can reduce path lengths and the size of the state space to a more tractable size. Our approach tracks how the state changes along a path in the state-transition graph. These variables' values continuously change causing states that would normally match to differ.

5.2.2.1 Identifying States

Applications have many states created at different locations in the code and on different paths. States only match when the execution is in the same location in the code since the thread state is included in the application state. There are too many states to compare them all to each other. The number of changes reported when comparing all states are also

Stellenbosch University https://scholar.sun.ac.za

too many for the user to identify variables causing the state explosion. To highlight changing variables, we therefore limit state comparison to subsequent states stored at the same location in the code and on the same path in the state-transition graph.

To mark states for comparison, the user inserts a `markState` statement into the source code:

```
StateComparator.markState(String tagName, int
        startAfter, int stopAfter);
```

The tag name distinguishes between different locations in the code to compare. Currently we only use a single location (mark-statement) in our examples. To reduce the reported changes and to ensure that the analysis terminates, the user can also specify the number of marked states after which to start recoding states and the number of marked states after which to stop the analysis. Each time the mark statement is reached, the current transition is ended and a new state is stored.

5.2.2.2 Serializing States

The tool serializes each marked state using a custom state serializer and caches it until the next marked state is reached. Our state serializer extends JPF's state serializer that is run each time a new state is reached by JPF. Our serializer caches all variable values of the state during the serialization phase when the entire state is traversed to calculate a hash value used for state matching. This saves execution time by reusing the serialization traversal of each state variable. Because the user can specify the number of marked states to record and when to start recording, only a few marked states are stored in order to detect changing variables. Therefore the tool does not greatly influence the performance of JPF.

Cached states created by JPF-Android's serializer store separate maps for static variables, stack frames and heap objects. A cached state stores the unique object/class id and names of variables mapped to their values. For stack frames it stores the local variables, the thread id and depth of the stack frame in order to compare variables across states. A variable can

Stellenbosch University https://scholar.sun.ac.za

either have a primitive value or store a reference to another object in the heap. References are stored as Strings starting with an '@' character to distinguish them from normal integers or Strings.

5.2.2.3 Comparing States

Two states are equal when all their variables (local, static and dynamic) match. Since marked states are expected to match, we are interested in the changes in their variables.

Variables with primitive values are compared using the normal Java equals operator. Variables with reference values, however, are more complex. To ensure that two objects are truly equal, their fields are compared recursively following references to other heap objects. Variable changes are recorded together with the path of objects from the root object to the changed object.

Corresponding local variables are matched across states using the unique id of its stack frame, the thread id to which the stack frame belongs and the name of the variable. Static variables can be compared directly using their class's id and the name of the variable, since a class is loaded only once.

Matching corresponding dynamic objects in the heap, however, is not directly possible. A simple example of this is the immutable `Integer` wrapper class in Java that cannot be modified. Each time the variable is changed, a new `Integer` object with a different reference is placed on the heap. Although the references differ, it might be the same variable in the program. To match dynamic objects we use an algorithm similar to the mark-and-sweep algorithm used during garbage collection to compare all reference objects in the heap and ensure each object is only visited once. Variables are compared by starting from the *roots* of the object graphs (the local and static variables) and then recursively comparing their fields until reaching primitive variables. In this way all referenced objects in the heap are compared. This graph of object references can contain cycles when objects reference each other. To avoid cycles during analysis, reference

Stellenbosch University https://scholar.sun.ac.za

objects are marked as visited when compared and skipped when reached again. The algorithm finds all changes in the object graph.

Changes are recorded and the new state cached replacing the previous state that is no longer needed. To enable the tool to work with multiple execution paths, the cached state is stored and backtracked with the application state.

5.2.2.4 Interpreting and Using Output

State Comparator prints the ids of compared states and all variable differences. It also prints the object trace from the root object to the mismatching variable in order to provide context to the variable. This allows the user to identify which variables keep on changing for each state comparison. Each marked state is printed to a file for manual inspection. If a field causing the state space explosion does not influence the property being checked, it can be removed from the state. Otherwise the field must be abstracted to a finite set of values to limit the behavior of the system to reduce the state space.

Variables can be abstracted in several ways using JPF. For example, they can be removed from the state using a `@FilterField` annotation. Filtering values from the state in this way removes the entire object hierarchy of the field from the state used for state matching. The variable still exists within the heap and its state is backtracked by the model checker, but it has no effect on state matching. The annotation also supports specifying a condition for when the annotated field should be filtered. The `@FilterField` annotation can be used on static and instance fields but not on local variables. The `@FilterFrame` annotation can filter method stack frames, their program counter and their local variables from the state. Annotations cannot always be added to libraries or application code. To overcome this issue, JPF-Android allows them to be configured in the JPF properties file. JPF also provides `AbstractionAdapters` so that fields can be abstracted using a custom method executed when the field is updated. In this case the user implements an adapter for each primitive

Stellenbosch University https://scholar.sun.ac.za

variable that needs abstraction: local, static or instance variable. When the value of the variable is set, the adapter is fired to ensure that its value stays within the bounds.

5.2.2.5 Examples

The usefulness of the StateComparator is illustrated using the binary tree example above and three other examples. Applications that have unbounded variables cause non-termination of the model checker. To detect these variables, the search is bounded using a search depth or by specifying the number of marked states after which to terminate the model checker. We also reduce all thread choices in the application to a single choice to reduce the detected changes using a custom thread scheduling strategy implemented by our tool. Lastly, a `markState` statement is added to the application at a point where the user expects state matching to occur.

A simple example with an unbounded number of states is shown in Listing 5.4. It contains an infinite while-loop that continues to increase the `iTest` local variable for each iteration of the loop. In this example, the model checker does not terminate since `iTest` will never have the same value and so state matching cannot occur.

```
1 public class SimpleExample {
2   public static void main (String[] args) {
3     int iTest = 1000;
4
5     while (true) {
6       iTest++;
7       System.out.println("iTest=" + iTest);
8       StateComparator.markState("TAG1");
9     }
10  }
11 }
```

Listing 5.4: SimpleExample application

To find the variable causing an unbounded state space using our tool, we inserted a `markState` statement on line 8. This statement breaks the

Stellenbosch University https://scholar.sun.ac.za

transition and marks the new state for comparison. We bound the search space to depth 10 in order to limit the results. The output of the tool is given in Listing 5.5. It shows the results of comparing marked states (0, 1), (1, 2), ..., (8, 9). These comparisons show that `iTest` is incremented for each state which causes the states to never match. We limit the `iTest` field to a maximum value of 1002 by extending JPF's `AbstractionAdapter` and re-run the application. Now the loop executed twice before the model checker terminated — the second time the marked state matched the previous marked state and the exploration was stopped.

```
=== COMPARING STATE 1 TO  STATE 0 ===
SimpleExample.main(...)V.iTest:  (1001 ==> 1002)

=== COMPARING STATE 2 TO  STATE 1 ===
SimpleExample.main(...)V.iTest:  (1002 ==> 1003)

...

=== COMPARING STATE 9 TO  STATE 8 ===
SimpleExample.main(...)V.iTest:  (1009 ==> 1010)
```

Listing 5.5: Tool output for SimpleExample

Our second application is the "oldclassic" example in jpf-core, inspired by a concurrency defect found on a space craft controller [63]. The application consists of two threads (`FirstTask` and `SecondTask`) that interact by exchanging events. An extract from the code is shown in Listing 5.6. The `SecondTask` starts by signaling `event1` that wakes up the `FirstTask` waiting on `event1`. When `FirstTask` is notified of `event1` it signals `event2` notifying the `SecondTask` of `event2`. A task can be notified of a new event before performing a costly wait operation. To optimize the application, each thread caches a copy of the event counter associated with the event on which it waits. If the event counter is increased before it starts to wait, it skips the waiting operation and instead processes the event.

The model checker never terminates on this example so we added a `markState` statement in the while-loop of the `FirstTask`. We expect

Stellenbosch University https://scholar.sun.ac.za


```
1 class Event {
2   int count = 0;
3   public synchronized void signal_event () {
4     count++; notifyAll();
5   }
6   public synchronized void wait_for_event () {
7     try { wait(); } catch (InterruptedException e) {}
8 }}
9
10 class FirstTask extends Thread {
11   Event event1; Event event2;
12   int   count = 0;
13   ...
14   @Override
15   public void run () {
16     count = event1.count;       // caches counter
17     while (true) {
18       StateComparator.markState("TAG1");
19
20       // waits if no event1 has been received
21       if (count == event1.count) {
22         event1.wait_for_event();
23       }
24       count = event1.count;
25       event2.signal_event(); // updates event2.count
26 }}}
27
28 class SecondTask extends Thread {... }
```

Listing 5.6: "oldclassic" example of jpf-core

states to match after a few iterations of this loop since no new behavior will be explored. The State Comparator detects four variables changing for each state comparison: the count variables. The changes recorded for the last two states are shown in Listing 5.7. These variables can be bounded to a maximum value, in the same way as the previous example, in order to enable state matching.

To detect the unbounded `modCount` variable in the binary tree example, we replace Line 8 with a `markState` statement. To bound the application we set a search depth of 10 and run the application through JPF with the State Comparator enabled. The changes detected by comparing the first two states are given in Listing 5.8. Here we can see that the `modCount` field of the binary tree object, defined as a local variable in the main method, is incremented by two for each loop iteration (once for adding an integer value and once for removing it). To reduce the state

Stellenbosch University https://scholar.sun.ac.za


```
FirstTask.count:  (11 ==> 12)
Object trace:
@163 object FirstTask, mFields={count=12, event1=@15e, event2=@15f
    ,...}
@1   frame FirstTask.run(...) locals={this=@163}

Event.count:  (11 ==> 12)
Object trace:
@15f object Event mFields={count=12}
@163 object FirstTask mFields={count=12, event1=@15e, event2=@15f
    ,...}
@1   frame FirstTask.run(...) locals={this=@163}

SecondTask.count:  (11 ==> 12)
Object trace:
@175 object SecondTask mFields={count=12, event1=@15e, event2=@15f
    ,...}
@1   frame SecondTask.run(...) locals={this=@175}

Event.count:  (12 ==> 13)
Object trace:
@15e object Event mFields={count=13}
@1   frame oldclassic.main(...) locals={args=@bb, new_event1=@15e,
    new_event2=@15f, task1=@163, task2=@175}
```

Listing 5.7: Changes detected for the last state comparison in oldclassic example

space we use JPF's `@FilterField` annotation to remove this field from state matching. When run again, the application state matches after the second iteration of the loop.

```
==== COMPARING STATE 1 TO  STATE 0 ====
BTree.modCount:  (2 ==> 4)
Object trace:
@15b object BTree mFields={elements=@15f, modCount=4, size=0}
@1    frame BTree.main(...)V: locals:{args=@bb, bt=@15b}

=== COMPARING STATE 2 TO  STATE 1 ===
BTree.modCount:  (4 ==> 6)
Object trace:
@15b object BTree mFields={elements=@15f, modCount=6, size=0}
@1    frame BTree.main(...)V: locals={args=@bb, bt=@15b}
...
```

Listing 5.8: Changes detected for BinaryTree example

The last example is a RSSReader Android application displaying the RSS feed entries to the user. The environment of the application is mod-

Stellenbosch University https://scholar.sun.ac.za

eled in JPF-Android. JPF-Android always returns the same set of feed items when the update button is pressed. These items are shown in a list displaying the name and the elapsed time since an item was posted. We expect the application to match after a few presses of the update button, but instead the model checker does not terminate. To identify the problem we analyze a single path in the application and compare the application state after each update button press using State Comparator. The changes detected for each state comparison are shown in Listing 5.9. We see that the `char[]` representing the text in the `mText` field of the `TextField` object is changing continuously. On further inspection we found that this `TextField` stores the elapsed time since the feed item was posted and thus will never be the same since the current time changes — even in JPF. We excluded this field from the state using a `@FilterField` annotation. Afterwards the application state matched after two presses of the update button.

```
mList:  [-,1,4,7,0,0,6,2,1,9,5,5,7,3,]
==> [-,1,4,7,0,0,6,2,1,9,6,9,1,8,]

Object trace:
@5d0a object char[] mList=[-,1,4,7,0,0,6,2,1,9,6,9,1,8,]
@5d09 object java.lang.String, mFields={value=@5d0a,...}
@5cb8 object android.widget.TextView, mFields={mText=@5d09,...}
...
```

Listing 5.9: Results for RSSReader example

These examples show how our tool can detect state changes and report them to the user. The last example highlights the usefulness of the tool in identifying unbounded variables in a large system that contain hundreds of variables.

5.2.2.6 Discussion

Previous work has been done on abstracting the environment of Java applications for analysis purposes [61, 38]. Our work focuses only on

Stellenbosch University https://scholar.sun.ac.za

detecting and bounding fields causing an infinite or too large state space. These fields are hard-to-find in large systems with thousands of variables.

VarTracker, a listener in jpf-core's [12] `gov.nasa.jpf.listener` package, counts the number of states for which fields, local variables and static variables change. Variables that change often can indicate that they are unbounded or have too many possible values. Although this tool can identify that `iTest`, the local variable in the SimpleExample (Listing 5.10), is changing, it cannot distinguish when a variable changes back to previous value — not hindering state matching. If we assign `iTest = (iTest + 1) mod 3`, for example, the same changes will be detected for each iteration if state matching does not occur.

```
change     variable
---------------------------------------
1000   SimpleExample.iTest
1      sun.misc.Unsafe.theUnsafe
1      SimpleExample.main([Ljava/lang/String;)V.se
...
```

Listing 5.10: Results of VarTracker for SimpleExample

Changes to fields are recorded for all instances of a class. If the application contains many of the same objects, the changes will accumulate fast for these fields, even though they may be bounded variables. Lastly, the larger the program is, the more variables change continuously which makes it hard to distinguish which variables should be bounded — especially when variables depend on each other. In the case of the binary tree, for example, the tool detects that `modCount` is modified, but it also detects that `size` and many other variables also change (see Listing 5.11).

False positives can be reported by State Comparator when the application contains variables preventing state matching as well as variables alternating between a set of values that would normally allow state matching. Additionally, it cannot detect indirect dependencies between variables. Abstracting one of the detected variable in this case may lead to multiple variable abstractions.

Stellenbosch University https://scholar.sun.ac.za


```
change    variable
--------------------------------------
10    BTree.remove(Ljava/lang/Object;)V.ii
10    BTree.ensureCapacity(I)V.ii
10    BTree.modCount
10    BTree.size
10    BTree.remove(Ljava/lang/Object;)V.found
 1    sun.misc.Unsafe.theUnsafe
 ...
```

Listing 5.11: Results of VarTracker for BinaryTree

5.3 Listener Extensions

JPF listeners have access to the application and the JPF VM's state. They are not part of the application state and are not backtracked by default. JPF-Android implements listeners to track the execution of the application and to record discovered event sequences. It also provides its own property specification mechanism *Checklists* to ensure the application's execution meets requirements set by the user.

5.3.1 Coverage Calculation

JPF-Android's goal is to improve the code coverage of traditional dynamic analysis tools. In order to perform a fair coverage comparison, JPF-Android includes a listener to calculate coverage in the same way as the other dynamic tools using EMMA [2]. The listener calculates the code coverage across all paths explored in the application. Android applications are instrumented using EMMA during compilation. The Android Ant/Gradle build scripts include options to build and instrument the Java source code using EMMA before converting it to DALVIK byte-code.

During instrumentation, EMMA inserts fields into classes to collect the coverage data for each class, method and basic block. JPF-Android excludes these fields from state matching but they are backtracked together with the application to ensure the correct coverage is recorded. These fields are registered in the `RT.java` class in the EMMA library and dumped to a file at the end of each path. These files are merged to find

Stellenbosch University https://scholar.sun.ac.za

the cumulative coverage across all paths.

5.3.2 Recording Event Sequences

JPF-Android generates a set of events to fire non-deterministically each time the application is idle. These events form event sequences which JPF-Android stores as an *event tree*. The nodes of the tree are processed events and the edges connect events executed after each other to form event sequences. The root of the tree is the fist event starting the application. The leaves of the tree are events after which the exploration of the event sequence was stopped due to state matching, no more available events were detected or a depth bound was reached.

The tree keeps a pointer (cursor) to the node of the current event processed by the framework model. When a property violation is found in an application, the event tree enables JPF-Android to trace back the sequence of events leading to the error using the current path. But if no errors are found, it is useful to retrieve statistics of the sequences explored by JPF-Android. The event tree wrapper listener stores the event tree as well as a list of all unique events fired for each Activity (can be used for script writing). It prints out a summary of the number of detected event sequences and a histogram of the lengths of the sequences. Since the event tree is stored in a listener, it is excluded from the application state. We do however backtrack the pointer to the current event in the tree to ensure it correctly records event sequences.

The detected sequences can be refined and reused to run sequences on JPF-Android or on an actual device to verify that they exist. This allows users to verify errors detected by JPF-Android on an actual device. We developed a basic tool to enable JPF-Android event sequences to be executed on the Android emulator. The tool serializes event sequences and can execute an event sequence on the emulator using a custom python script. The script is built on a low level monkeyrunner [36] API that allows scripts to search for specific views and fire UI and system events. Our script makes use of a telnet connection to the running device to set the

Stellenbosch University https://scholar.sun.ac.za

state of certain services such as the connection status. The main problem encountered by this approach is the limitation to configure the device due to built-in security of the device. Shared preferences and the state of the databases, for example, cannot be set from outside of the app.

5.3.3 Property Specification with Checklists

Model checkers detect errors in concurrent applications using Linear Temporal Logic (LTL) property specifications verified for each path, across all execution threads. Using LTL formulas to define property specifications has the disadvantage of easily becoming complex for certain properties [29].

Android applications have a multi-threaded, event-driven and asynchronous message-passing design which makes it hard to specify properties using only LTL formulas. JPF-Android introduces Checklists as succinct logical specifications designed to simplify verifying the execution of Android applications on JPF.

Checklists automatically verify the execution triggered by an event, during runtime, to ensure that the application executes as expected. Checklists are verified for each thread of execution compared to LTL formulas which are verified across all threads. In order to track execution in these multi-threaded, asynchronous applications, copies of the current Checklists and their states are passed to newly spawned threads and to threads during message passing.

Checklist are not designed to detect liveness properties, but to validate that applications follow requirement specifications. They cannot represent all property types specified with LTL or regular expression formulas. This functionality is sacrificed for the sake of clarity and simplicity of representation.

5.3.3.1 Checklists

Checklists consist of an ordered sequence of *Checkpoints* that represent methods invoked by specific threads. During execution, Checkpoints are

Stellenbosch University https://scholar.sun.ac.za

matched one-by-one, in order, to marked methods (Checkpoint methods) in the application code. Although a Checklist is matched in sequence, Checkpoint methods may be reached in the code that are not relevant to the Checklists. Methods that do not advance a Checklist are simply ignored.

⟨*checklist*⟩ = ⟨*id*⟩ ':' ⟨*guard*⟩ '=>' ⟨*checkpoints*⟩ ';'.
⟨*guard*⟩ = [⟨*checkpoint*⟩ { ',' ⟨*checkpoint*⟩ }].
⟨*checkpoints*⟩ = ⟨*checkpoint*⟩ [{ ',' ⟨*checkpoint*⟩ }].
⟨*checkpoint*⟩ = ['!'] ⟨*id*⟩.
⟨*id*⟩ = letter { letter | digit | '_' }.

Figure 5.1: EBNF for Checklists

Checklists are defined using the syntax in Figure 5.1. Checklist definitions consist of three parts. The *id* of a Checklist is used to uniquely identify the Checklist. A Checklist definition also contains a list of Checkpoints, separated into the *guard* and the *checkpoints* by the implication symbol ("=>"). The guard acts as a condition to identify the paths and threads for which to match the Checklist. If the Checklist is matched past the implication symbol, the Checklist reports violations; otherwise, the Checklist was not meant for this thread. A Checklist can have an empty guard, but, not an empty list of Checkpoints.

Each time the first positive Checkpoint or a negative Checkpoint preceding it is matched, a new instance of the Checklist is created. Multiple instances of a Checklist can be verified at the same time. All Checklist instances are added to the collection of active Checklists verified for an event.

A negative Checkpoint is indicated by placing a "!" symbol in front of the Checkpoint's name in the Checklist definition. A negative Checkpoint signifies that this Checkpoint may not be reached before the next positive Checkpoint is reached or before the end of the path.

Stellenbosch University https://scholar.sun.ac.za


```
1 class UpdaterThread extends Thread {
2
3   @Checkpoint(value = "runUpdate", threadname = "UpdaterThread")
4   public void run(){...}
5
6 }
```

Listing 5.12: Example of a Checkpoint annotation

```
1 [Mappings]
2 runUpdate : com.android.vdm.UpdaterThread.run()V, UpdaterThread;
```

Listing 5.13: Example of a Checkpoint Mapping

Checklist can be violated in two ways: when a negative Checkpoint is reached in its checkpoints or when all positive Checkpoints in its checkpoints are not reached before the path ends. When a violation occurs in the checkpoints of the Checklist, JPF-Android stops execution and outputs the violation details.

5.3.3.2 Defining Checkpoints in the Application

JPF-Android makes use of method calls to represent Checkpoints in the application code. A method can only be bound to a single Checkpoint. Java allows methods in the same class to have the same name but different parameters (polymorphism) and methods in different classes to have the same signature. To clearly distinguish between methods, Checkpoints can be created by either adding a Checkpoint Java annotation to the method (see Listing 5.12) or by mapping the full signature of a specific method to a Checkpoint in the Checklists definition file (see Listing 5.13). The mapping functionality is useful when the code of the SUT cannot be changed.

If a class extends another class, it inherits all methods of the superclass. Checkpoints are inherited together with the superclass's methods. If a class overrides methods from its superclass, the Checkpoint annotations on these methods are not inherited. The overridden method's Checkpoint will only fire if its supermethod is called.

Stellenbosch University https://scholar.sun.ac.za


```
update:
 !batteryLow, !WifiDown
  =>  runUpdate, displayResults;
```

Listing 5.14: Specifies that an update should happen when the battery is not low and the WiFi is connected.

Checkpoint definitions allow the user to specify the name of the thread that should reach the Checkpoint. It allows the Checklist to distinguish between, for example, the main thread calling a method which may also be called from another thread spawned from the main thread.

5.3.3.3 Negative Checkpoints

As discussed above, a Checklist's guard and checkpoints support negative Checkpoints. Although negative Checkpoints are useful in verifying that a specific event does not occur before or after another, their meaning becomes ambiguous in some cases. Let us look at such an example.

The Checklist in Listing 5.14 verifies that when an update is started, the Wifi is connected and the battery is not low, the update completes successfully by displaying the results. When a positive Checkpoint is preceded by one or more negative Checkpoints (shown in Listing 5.14), it means that neither of the negative Checkpoints may be reached before the next positive Checkpoint (the runUpdate Checkpoint in this case). The order of these negative Checkpoints is insignificant. The `!WifiDown` and `!batteryLow` Checkpoints are used to only select paths where the Wifi is configured to be connected and battery level to high. Therefore, if either the `WifiDown` or `batteryLow` Checkpoint is reached, this Checklist accepts and is removed from the list of active Checklists since it is not relevant for this path.

Stellenbosch University https://scholar.sun.ac.za

5.3.3.4 Implementing Checklists as Deterministic Finite Automata (DFA)

Checklist are implemented as DFAs. The guard and the checkpoints of the Checklist are represented by a single DFA. Each Checklist instance stores a pointer to the last Checkpoint matched. When a violation occurs, the Checklist instance checks whether the pointer has matched past the guard before reporting a violation. Checklists can be converted to DFAs by following the steps below.

1. Start the DFA as an accept state with a transition to itself for all Checkpoints in its alphabet.

2. For each positive Checkpoint, add a transition from the current state to a new state. Edit the transition from the former state to itself to exclude this Checkpoint. Change the former state to a normal state and update the new state to an accept state. Add a transition from the new state to itself for all Checkpoints in its alphabet

3. For each negative Checkpoint, change the Checkpoint to a positive Checkpoint. If the Checkpoint is in the guard, add a transition from the current state to an accept state for this positive Checkpoint. If the Checkpoint is in the checkpoints of the Checklist, add a transition from the current state to a non-accept, sink state for the positive Checkpoint. Edit the transition of the former state to itself to exclude this Checkpoint.

Four basic Checklist examples are given in Figure 5.2.

The DFA can either reject or accept the input depending on its current state. If it reaches a sink state or the path ends without reaching all positive checkpoints, the Checklist is reported as violated. If a Checklist is in its guard, the violation is ignored since the guard is only used to select a specific path and execution thread.

Stellenbosch University https://scholar.sun.ac.za

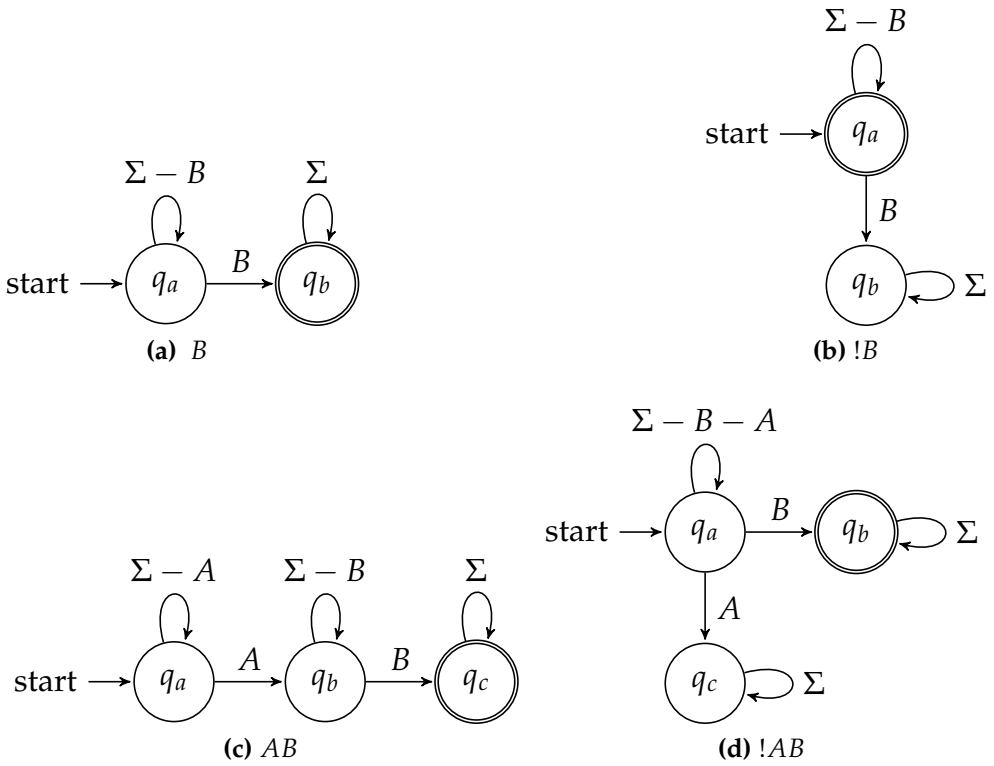


Figure 5.2: Basic Checklists translated to DFA

5.3.3.5 Examples of Checklists as DFAs

These steps are used to construct a DFA that represents the Checklist in Listing 5.14. The DFA is shown in Figure 5.3.

A new instance of the update Checklist is created when the `wifiDown`, `batteryLow` or the `runUpdate` Checkpoint is reached in the application code. If the `wifiDown` Checkpoint is reached, the Checklist moves to an accept. Since the `runUpdate` Checkpoint can never be reached from there, the Checklist instance cannot be matched passed the guard so it is discarded. If the `runUpdate` Checkpoint is matched, the Checklist instance continues to match the `displayResults` Checkpoint after which it accepts.

In another example, presented in Listing 5.15, the guard is empty and the negative Checkpoint is part of the checkpoints of the Checklist and

Stellenbosch University https://scholar.sun.ac.za

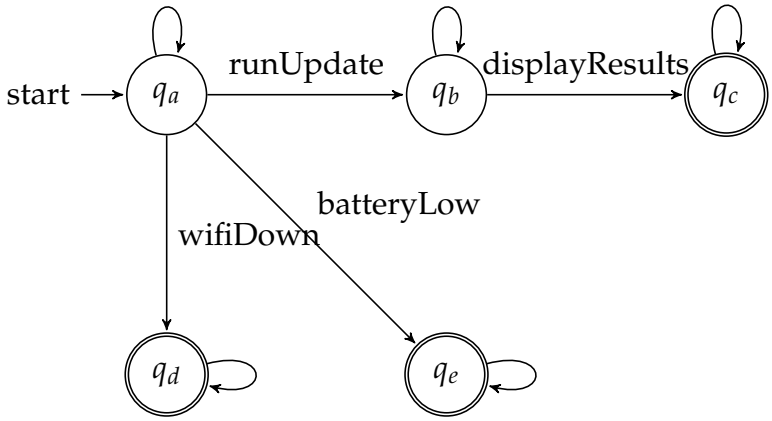


Figure 5.3: DFA representing Checklist update

not in the guard of the Checklist. A Checklist instance is only created when the `update` Checkpoint is matched. This Checklist verifies that a `cancelUpdate` Checkpoint is never reached after an `update` and before a `runUpdate` Checkpoint. The Checklist will fail if the `cancelUpdate` Checkpoint is reached after the `update` because the guard is empty. In Figure 5.4 we can see that when the `cancelUpdate` Checkpoint is matched, the DFA goes into a sink state from which it will never be able to transition to an accept state.

```
runUpdate3:
    update, !cancelupdate, runUpdate;
```

Listing 5.15: Defining runUpdate3 Checklists

5.3.3.6 Multi-Threaded Applications

Android applications are multi-threaded. When a new thread is started, the execution splits into two concurrently running threads. JPF-Android creates a copy of all Checklist instances for each thread. It records the thread on which a Checklist instance is created and on which Checkpoints are reached. Checklists can only be advanced by Checkpoints reached on the same thread (or on one of its offspring). Android applications also makes use of message passing between threads to make asynchronous

Stellenbosch University https://scholar.sun.ac.za

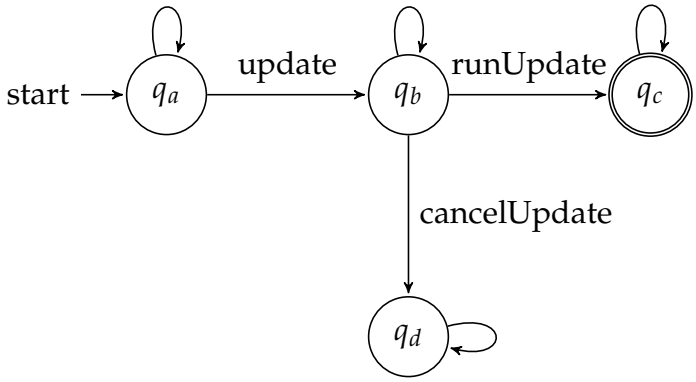


Figure 5.4: Illustration of Checklist runUpdate3

calls to application components and threads. For this reason JPF-Android also copies Checklists across threads together with messages to allow following the execution across threads. Tracking the execution in this way is computationally expensive. Therefore Checklists only verify the execution triggered by a single event generated by the event producer.

Figure 5.5 illustrates the execution across threads when the update button is pressed until the display is updated. The two threads are displayed on the y-axis (Main thread and Update thread) and the execution time is shown on the x-axis. The black lines show the application execution as time progresses. The black points indicate when Checkpoints are reached by a thread. We can see that when the Update thread is started by the Main thread (dotted arrow pointing upwards) the execution splits. The same happens when a message is posted asynchronously back to the Main thread from the Update thread (indicated by a dotted arrow pointing downwards). The blue line indicates how the Checkpoints are matched on different threads.

5.3.3.7 RSSFeedReader Example

The RSSFeedReader app downloads and stores RSS feed items in a database. The most recent items are displayed in a list combining items of all the RSS feeds. When an item in the list is selected, a web page is shown containing the item's content.

Stellenbosch University https://scholar.sun.ac.za

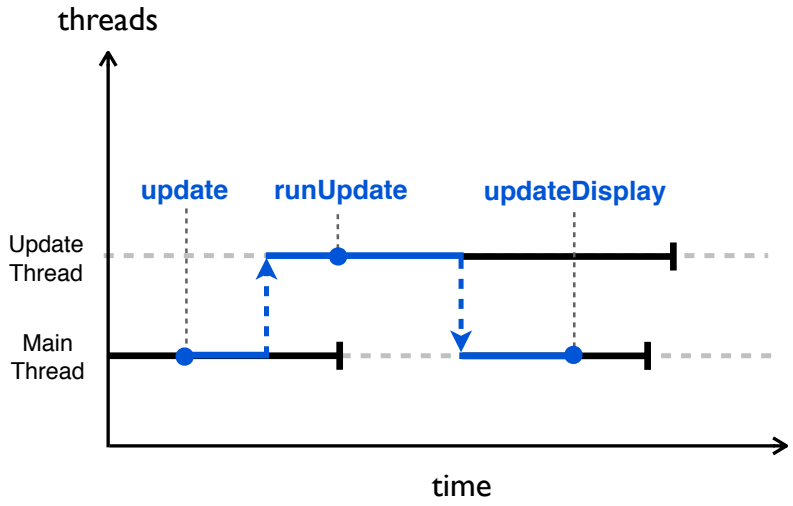


Figure 5.5: Execution flow over two threads

JPF-Android aims to provide the user with as much freedom as possible to configure external input to the application. This input can be non-deterministically changed by using an Any script element. The network connection is modeled by allowing the user to associate a URL with a filename containing the data sent over the network from within the input script. In Listing 5.16, lines 2–3 a specific predefined Intent object, called an `urlInputStreamIntent`, is constructed and sent to the system from the script. When a network connection is made to this URL, JPF-Android returns the specified file's contents. This input stream is then sent to the XML parser to be parsed.

When the user clicks on the refresh button in the TimelineActivity, the application starts a new thread to retrieve the newest items from each registered RSS feed and updates the list to reflect these changes. For the sake of simplicity, downloading updates are only allowed over an active WiFi connection. If the user clicks the refresh button and either the WiFi is not connected, the battery is too low or an update is already running, the application should not attempt to update (refresh) the feeds but notify the user of the error. To verify that the application correctly handles each of these situations, the script in Listing 5.16 is used to simulate each of

Stellenbosch University https://scholar.sun.ac.za


```
1 SECTION default {
2     @urlInputStreamIntent.putString("url","http://feed.rss")
3     @urlInputStreamIntent.putString("file","src/input.rss")
4
5     sendBroadcast(@urlInputStreamIntent)
6
7     @startIntent.setComponent("TimelineActivity")
8     startActivity(@startIntent)
9 }
10
11 SECTION TimelineActivity {
12     ANY{
13         GROUP{
14             device.setWifi("ON")
15             device.setBattery("100%")
16         },
17         GROUP {
18             device.setWifi("OFF")
19             device.setBattery("100%")
20         },
21         GROUP {
22             device.setWifi("ON")
23             device.setBattery("1%")
24         }
25     }
26     $buttonRefresh.onClick()
27 }
```

Listing 5.16: Input Script

these situations non-deterministically. The Checklists in Listing 5.17 are registered to verify this behavior.

A Checklist can be violated in two ways: firstly, when a negative Checkpoint is reached or when all Checkpoints in a checkpoints of the Checklist are not reached by the end of a path (if the guard was matched). Checklists are matched during runtime and violating Checklists are reported at the end of execution. To illustrate how Checklists are reported, two violations have been introduced into the RSSFeedReader application. Figures 5.6 and 5.7 display the violations as reported by JPF-Android. In Figure 5.6 the `getFeedUpdates` Checklist failed because the `storeInDB` Checkpoint was not being reached. In Figure 5.7 the application tried to update although the WiFi was not connected, which violated the `!runUpdate` Checkpoint in the `updateWifiDown` Checklist.

Stellenbosch University https://scholar.sun.ac.za


```
1   // alreadyRunning, WifiDown or batteryLow not allowed just
2   // before running the update
3   update:
4         update, !alreadyRunning,
5         !WifiDown, !batteryLow
6     =>  runUpdate;
7
8   // after parsing the Feed the items must be stored in the DB,
9   // re-loaded from the DB and the listview updated
10  getFeedUpdates:
11    =>  parseFeed   storeInDB, loadFromDB, updateListView;
12
13  // if an update is attempted while the WiFi is down, the
14  // update must not run but be canceled an the user notified.
15  updateWifiDown:
16        update, WifiDown
17    =>  !runUpdate, cancelUpdate, notifyWifiDown;
18
19  // if an update is attempted while the battery is low, the
20  // update must not run but be canceled an the user notified.
21  updateBatteryLow:
22        update, batteryLow
23    =>  !runUpdate, cancelUpdate, notifyBatteryLow;
24
25  // if an update is attempted while running an update the
26  // update must not run but be canceled an the user notified.
27  updateRunning:
28        update, alreadyRunning
29    =>  !runUpdate, cancelUpdate, notifyAlreadyRunning;
```

Listing 5.17: Checklist Definitions

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

Figure 5.6: Violation 1

Stellenbosch University https://scholar.sun.ac.za


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

Figure 5.7: Violation 2

Stellenbosch University https://scholar.sun.ac.za

Chapter 6

Evaluation

We evaluate the effectiveness and efficiency of our approach by analyzing a set of representative applications. Most of the apps selected for the benchmark come from the paper entitled "Automated Test Input Generation for Android: Are We There Yet?" [28]. Only a subset of these apps are used since it is time consuming to understand the behavior of each app to optimize models and to ensure the app is running as expected. This is also necessary to verify crashes/errors. The Calculator and RSSReader apps are test applications we developed to test a range of functionality provided by the Android framework.

The apps used for the experiments are selected to have different attributes, dependencies and require different configurations from their environment. K9Mail, for example, is a very large app whereas Calculator is a very small app. The apps have different number and types of external dependencies. Some apps require very specific user input or system configurations whereas other apps only require very basic input. The RSSReader app, for example, requires very little and simple user input but relies on complex environment configurations to achieve high code coverage. SyncMyPix, on the other hand, requires the user to log in with a valid Facebook account before exposing the bulk of the app's behavior. We also included apps such as tippy tipper that are heavily dependent on preference values to achieve good coverage whereas the

Stellenbosch University https://scholar.sun.ac.za

Table 6.1: The apps used for evaluation. Their size is given in LOC and number of components: Activity (A), Service (S), BroadcastReceiver (BR), ContentProvider (CP). The number of models and events (E) show the size of their environment.

	Version	LOC	A	S	BR	CP	Models	E
Calculator	2.0	114	2	0	0	0	2	40
AutoAnswer	1.5	140	1	1	2	0	8	10
RMP	1	315	1	1	1	0	4	11
AnyCut	0.5	692	4	0	0	0	5	18
RSSReader	2.0	774	2	1	1	0	6	6
aGrep	0.2.1	1505	5	0	0	0	3	24
Tippy Tipper	1.1.3	1771	5	0	0	0	10	52
PasswordMaker	1.1.7	2310	3	0	0	0	8	30
SyncMyPix	0.15	4081	8	1	2	1	53	31
Keepassdroid	1.9.8	4972	14	1	3	0	56	69
Ringdroid	2.6	5394	3	0	0	0	37	40
K9mail	3.512	47931	25	5	5	2	114	33

Calculator app does not make use of preferences. The same goes for using a database or cursor and relying on different types of files on the file system. The applications vary in terms of Lines of Code (LOC) and number of components.

Table 6.1 shows the number of application-specific models created per app and the number of unique events used to explore each app using JPF-Android. These metrics give an indication of the size and complexity of the applications. Tippy Tipper, K9Mail and Keepassdroid, for example, respond to many events, so they have many possible event sequences.

For the first experiment we compare the statement coverage and the number of paths explored by JPF-Android to two state-of-the-art dynamic analysis tools for Android: Dynodroid [52] and Sapienz [55]. Dynodroid and Sapienz currently achieve the best results in terms of code coverage for automated input generation dynamic analysis tools.

Experiment 2 compares the different optimizations implemented by JPF-Android and the effect they have on the coverage and the number of paths and states explored.

Stellenbosch University https://scholar.sun.ac.za

Statement code coverage records what portion of the application statements are executed during analysis. It does not take into account that defects and risk are not distributed uniformly across the application code or tell us anything about the correctness of the analysis [37], but it gives us a measure of the confidence we can put in the analysis results. If the code coverage obtained by an analysis is low, the tool's results and run times are unreliable and skewed since only a small portion of the application code was analyzed. In these experiments we measure the statement coverage of the application code using EMMA [2]. All external libraries are excluded from coverage calculation since we assume these can be analyzed separately. We also excluded code injected into apps by the dynamic tools to dump the code coverage during runtime because JPF-Android's EMMA listener does not require this code.

6.1 Experiment 1: Code Coverage

We evaluate Dynodroid and Sapienz by running the applications on the emulator:

Dynodroid makes use of a biased-random input generation approach, firing events more often if they are relevant in more contexts. It employs a single, customized emulator to execute one long event sequence per run, either bounded by time or the number of fired events. Since it makes use of a random input generation approach, the tool needs to be run multiple times on each app and the results merged to collect more accurate coverage results obtainable by the tool. Therefore, we repeated the experiment as performed by Choudhary *et al.* [28] running each app on the tool for ten runs of an hour which should be more that sufficient since their paper showed that the apps reached maximum code coverage within five to ten minutes.

Sapienz uses a multi-objective, Pareto-optimal, genetic search-based algorithm that systematically evolves test scripts to minimize event sequence length and maximize code coverage [55]. It generates a set of 50 test scripts which are evolved into new generations up to 100 times to

Stellenbosch University https://scholar.sun.ac.za

optimize their results. Sapienz employs multiple concurrently running emulators to exercise event sequences and collect the coverage data. We recreated the experiment as performed by Mao *et al.* [55] and ran each application on the tool for an hour (the same as Dynodroid) using the default setting of 5 emulators. The tool generates test suits and coverage reports for each test sequence which we merge to obtain the final coverage.

Before the applications can run on JPF-Android, application-specific models were generated using OCSEGen (using a combination of runtime and default values). In some cases these models needed to be optimized manually to enable the applications to run. [1] This process takes a day or two depending on the user's knowledge of the application and its complexity. The number of these application-specific models generated for each app is given in column "Models" in Table 6.1. Once set up, these models can be reused for future runs and can be adapted for JUnit Testing.

For this experiment we configure JPF-Android to:

- perform a heuristic search (limiting events to be executed only once per path),
- bound event sequences to length 20 and limit the search depth to 1000 states (so that the analysis is guaranteed to complete), and
- explore all environment choices only once (the first time they are reached),
- use the default JPF-Android thread policy that only explores a single thread interleaving.

Table 6.2 shows the coverage achieved for each application. The highest coverage achieved for each app is highlighted.

Ten of the apps in Table 6.2 were previously analyzed by Dynodroid and Sapienz in papers by Choudhary *et al.* [28] and Mao *et al.* [55]. For eight of the ten apps we achieved higher coverage in this experiment using

[1] The models generated for the applications in these experiments are available at `https://bitbucket.org/heila/jpf-android-examples/src`.

Stellenbosch University https://scholar.sun.ac.za

Table 6.2: Shows the statement coverage for Sapienz (S), Dynodroid (D) and JPF-Android (J). For JPF-Android we also show the number of new states (#S), paths (#P) and environment choice points (#C) explored and the runtime (t).

	Coverage %			JPF-Android			
	S	D	J	#S	#P	#C	t
Calculator	**97**	**97**	96	17	128	3	4s
AutoAnswer	31	67	**98**	57	87	37	4s
RMP	62	93	**95**	372	484	65	14s
AnyCut	72	69	**88**	98	74	21	4s
RSSReader	34	-	**87**	36	20	9	2s
aGrep	63	**67**	54	21	322	0	17s
Tippy Tipper	86	79	**88**	1182	2947	363	2s
PasswordMaker	**80**	48	66	156k	178k	163	3h29m43s
SyncMyPix	20	20	**45**	12k	4518	516	6m49s
Keepassdroid	15	22	**47**	31k	67k	95k	2h42m15s
Ringdroid	44	-	**53**	179k	150k	2590	2h28m38s
K9mail	6	4	**17**	1290	1222	0	5m54s

Dynodroid when compared to previous papers. Two of the applications achieved nearly similar coverage. For Sapienz we also saw an increase in coverage in seven of the apps from its paper's results while the other three apps obtained very similar coverage results. We attribute this coverage increase to the fact that we excluded external libraries and instrumentation code from code coverage calculation.

In this experiment Dynodroid and Sapienz achieve low coverage with Sapienz obtaining less that 80% coverage on nine of the twelfth apps it analyzed, while Dynodroid obtained less that 80% coverage for seven of the ten apps it could analyze. This is due to the fact that they both run on an emulator with restrictions. Sapienz runs on a newer emulator with more behavior which can achieve better coverage in some cases. Sapienz could analyze all the applications where Dynodroid crashed on RSSReader and Ringdroid. JPF-Android, however, obtained code coverage of above 85% in six of the twelfth apps. It also achieved higher or similar coverage in all but two applications: PasswordMaker and aGrep. aGrep contains a

Stellenbosch University https://scholar.sun.ac.za

large amount of GUI code to construct custom widgets. Since JPF-Android does not model GUI measurements and drawing, it could not cover this code. PasswordMaker requires firing an event multiple times which was not enabled in this run to limit the search space.

Sapienz explored event sequences of length between 20 and 500 whereas Dynodroid explored event sequences with between 197 and 754 events per path with an average of 400 events per app per run. JPF-Android explores sequences systematically bounding event sequences to length 20. It analyzed all but one application, Keepassdroid, before reaching this bound. On further inspection we saw that this application has 14 Activities and 69 unique events were fired to exercise the application so sequences of length 20 were just not sufficient to reach all application behavior. Please note that since JPF-Android does not use events to change the device configuration but non-deterministically returns a subset of the configurations, its event sequences are shorter.

Since JPF-Android non-deterministically explores different environment configurations, we count the number of paths explored by JPF-Android. Each path encodes a single event sequences and environment configuration (since environment data choices are only made once.) The total number of paths explored by JPF-Android is given in Column 5 of Table 6.2. Dynodroid and Sapienz explore event sequences deterministically, updating the configuration of the emulator using events. Dynodroid explored 10 event sequences (one per emulator) and Sapienz explored between 38 and 327 event sequences per app. JPF-Android explores between 20 and 178k paths through the application. Although these dynamic tools explore only a small number of paths on paper, one cannot directly compare these longer sequences to the shorter paths of JPF-Android. The shorter paths might be subsumed under the longer sequences.

JPF-Android's analysis time was between 2s and 7 minutes for nine of the apps, but increased to over two hours for the three more complex and larger apps Ringdroid, Keepassdroid and PasswordMaker. The problem with all three tools is that there is no way to know when to kill the analysis. JPF-Android stops when all paths have been explored and the system

Stellenbosch University https://scholar.sun.ac.za

is finite state. The main problem comes in when JPF-Android does not complete in a given time limit. In this case the user must bound the analysis using either event sequence length, search depth or runtime. JPF-Android, however, completes its analysis without reaching the bounds of event sequence length of 20 or search depth 1000 for all but one application in this experiment.

6.2 Experiment 2: Optimizations

This experiment evaluates the different optimizations implemented by JPF-Android to decrease the analysis size without influencing the code coverage. In order to observe the influence of each optimization, we analyze the applications on JPF-Android using a configuration similar to Experiment 1:

- perform a heuristic search (limiting events to only be executed once per path),
- bound event sequences at length 4 and limit the search depth to 1000 states (to ensure the analysis terminates),
- explore all environment choices only once (the first time they are reached),
- use the default JPF-Android thread policy that only explores a single thread interleaving.

We then disable each of the optimizations and compare the number of states, number of paths, the code coverage and the runtime for each run. Since we expect that disabling the optimizations increases the analysis size, we bound the event sequences to length 4 in order to complete the runs on JPF-Android. Table 6.3 shows the results.

Table 6.3: Shows the results of using a heuristic event generator, default event generator, no state matching and no runtime values for event sequences of length 4.

	Heuristic				Default				No State Matching				No RV
	# S	# P	t (s)	C %	# S	# P	t (s)	C %	# S	# P	t (s)	C %	C %
Calculator	20	97	3	96	20	120	4	96	10294	9414	1m18	96	96
AutoAnswer	54	71	4	98	55	72	4	98	213539	74752	20m23	98	22
RMP	265	265	8	93	329	412	10	95	2335	1680	29	93	93
AnyCut	23	19	2	83	23	19	2	87	34	19	2	83	87
RSSReader	36	19	2	87	52	25	3	87	140	41	4	87	39
aGrep	115	107	6	54	125	145	7	56	493	346	11	54	49
TippyTipper	341	491	19	88	342	506	22	88	97903	64518	41m11	89	84
PasswordMaker	55	58	5	64	54	62	5	65	t/o	t/o	t/o	t/o	65
SyncMyPix	545	155	15	45	592	178	17	45	2571916	t/o	3h02m06	t/o	33
Keepassdroid	140	131	17	33	210	205	25	33	7191	4520	9m11	33	25
Ringdroid	1151	1000	46	53	1283	1196	58	53	t/o	t/o	t/o	t/o	28
K9mail	91	58	18	17	112	83	25	17	t/o	t/o	t/o	t/o	17

Stellenbosch University https://scholar.sun.ac.za

Stellenbosch University https://scholar.sun.ac.za

6.2.1 Event Generation

The "Heuristic" column shows the results of using JPF-Android's heuristic event generator (also used in Experiment 1), but bounding the analysis to event sequences of maximum length 4 instead of 20. We found that for most applications the coverage is very close to that of Experiment 1. This means that for most apps all entry-points are fired at least once using sequences of length 4, although all paths of the application are not necessarily explored at this point. We can also see that applications that respond to many types of events require longer event sequences. Keepassdroid has 14 Activities and responds to 69 events so its coverage deceased by 14% when shortening the sequences to length 4.

For the "Default" column we used the default event generator to see the impact on coverage when allowing events to occur multiple times within an event sequence. The default event generator in JPF-Android fires all possible events whereas the heuristic event generator fires each event once per path. The results show that the number of states and paths increase while improving the coverage of the apps by less than 5% each.

6.2.2 State Matching

State matching bounds the state space by stopping exploration in a path when it reaches a previously visited state. The column entitled "No state matching" shows the results of disabling state matching. The number of states explode in this case with three of the twelfth apps not completing and hitting the state depth of 1000 within five hours. These have been marked as "t/o" (timed out). This shows how large the actual state space of these applications are and the immense reduction in the number of states when using state matching.

6.2.3 Runtime Values

JPF-Android makes use of runtime values in models for preferences and cursors as well as for parameters for generated events. In the "No RV" col-

Stellenbosch University https://scholar.sun.ac.za

umn we show the decrease in coverage when runtime values are disabled. The coverage reduces drastically for apps utilizing these runtime values such as AutoAnswer, RSSReader and Ringdroid. Additionally removing runtime values and using default values caused 8 of the 12 apps to crash during the analysis.

6.3 Discussion

6.3.1 Coverage

Although JPF-Android achieved higher coverage for most of the applications, it could not fully cover all application code in Experiment 1. The challenges JPF-Android as well as the other tools face include:

Dead Code All applications contain dead code. Dead code can be left over from a previous version of the application and is now unused. Covering this code is impossible for all tools including JPF-Android.

Exceptions Java applications contain many try-catch blocks. JPF-Android could be modeled to throw exceptions and return invalid values to cover all exceptional code. Enabling this behavior while verifying the entire application, however, explodes the already large state space.

File system Certain applications such as Keepassdroid, Password-Maker and Ringdroid execute differently depending on the files on the file system. JPF-Android does not backtrack file creation, deletion or state since it requires too much memory and storage to keep a copy of the files for each state. JPF-Android does, however, allow the application to create and delete files to enable this code. Although writing to and reading from the file system can be modeled, apps often expect very specific files and file content which becomes complex to model to enable different application behavior.

Database Backtracking the state of a DB is not feasible when using JPF. We found that stubs returning runtime and default values for cursors are simple to generate and achieve acceptable coverage for many applications.

Stellenbosch University https://scholar.sun.ac.za

When a cursor is used to traverse different data sets, however, runtime values become less effective.

GUI Measurements and Drawing JPF-Android abstracts the GUI heavily and does not fire `onDraw` or `onMeasure` callbacks. This limits the coverage of custom views that utilize these methods or rely on the values of the physical dimensions of the applications.

Thread Scheduling JPF-Android only explores a single thread interleaving. The main problem with this approach is that certain applications like SyncMyPix execute different code depending on when external threads finish. When run a few times with random thread choices SyncMyPix achieves a coverage of 56% compared to its 45% reported for the default JPF-Android thread policy.

Event Generation The heuristics we use to reduce the event sequences are not always effective enough to reach all paths in the application. Also, we may not fire events with all possible parameters since they have not been observed during runtime.

6.3.2 Comparison to Other Tools

An important consideration when comparing dynamic analysis tools is how to compare the events/event sequences generated by the tools. In the case of Dynodroid, the longer the tool runs, the longer sequences are generated and the number of runs determine the number of sequences explored. Sapienz limits the event sequences generated to between 20-500 events. Another consideration before comparing event sequences is whether device configuration changes are counted as events. In JPF-Android the number of event sequences and number of paths explored differs since a sequence can non-deterministically be executed for different environment configurations. For this reason we report the number of paths explored by JPF-Android. Dynodroid and Sapienz change the environment configuration deterministically by firing events. This leads to many unnecessary events being fired that do not influence the execution triggered by the event sequence.

Stellenbosch University https://scholar.sun.ac.za

6.3.3 Environment Modeling

The environment model enables JPF-Android to explore event sequences and environment configurations in a controlled environment and enable behavior difficult or impossible to trigger on a device or emulator. We manually checked component life-cycle method traces of applications running on the tool against the traces generated on the emulator. We do not, however, try to prove the soundness or completeness of the models and leave this as future work.

Other dynamic analysis tools run Android applications on an emulator/device. The configuration and capabilities of the device (set implicitly in some cases) determine what application functionality can be triggered. For example, test devices connected to the Internet can make connections to web services. Devices can have specific files on their file system, an emulated/actual SD Card inserted, the language set to English and run a specific Android version. Dynodroid and Sapienz also both specifically add a music file to the file system of test devices to enable code of music player/audio applications to execute. They also choose String/integer values to use for input widgets because empty values can crash applications.

In order to run the application on JPF-Android we perform automatic (runtime values) or manual optimizations to models to enable Android application code. In cases such as the SyncMyPix application we manually optimization to get past the Facebook Login screen. In this case the interaction with the Facebook library was too complex to use runtime values. Optimizations were also made to the Url class for the RSSReader application to use a specific file as input instead of connecting to an on-line service to retrieve the XML of RSS updates. These optimizations might not be necessary on a physical device/emulator since the environment already supports a specific configuration. In our case modeling allow us to achieve better coverage for Android applications. Additionally, creating a model that can be reused can be much simpler to test different environment configurations than, for example, running an application on

Stellenbosch University https://scholar.sun.ac.za

devices with different Android versions, files on the file system or even states of external web services. Experiment 2 measures the effectiveness of using these types of optimizations by disabling all runtime values in generated and manual optimization of models.

6.3.4 Bug Reporting

Although JPF-Android did not report any bugs, in two of the applications, SyncMyPix and AnyCut, it reported that application threads were still waiting (blocked) and never killed by the application before terminating. Sapienz reported a unique crash in four of the twelve apps. In Ringdroid and PasswordMaker JPF-Android did not cover the lines of code where the exception occurred. The exception in K9mail was a crash in the framework code which is modeled by JPF-Android. The last error was found in Ringdroid where a cursor was accessed after it had already been closed. We stubbed the cursor in such a way that we could not detect this error.

Stellenbosch University https://scholar.sun.ac.za

Chapter 7

Conclusion

We conclude this thesis by presenting an overview of the project, results and contributions and by discussing limitations and future extensions of this work.

7.1 Verifying Android Applications

Android applications are hard to analyze due to their many entry-points and the number of configurations of their environment. Static analysis over-approximates the behavior of applications which can lead to false positives that are difficult to prove since they do not provide event sequences and environment configurations leading to errors. Dynamic analysis tools under-approximate the behavior of applications since they only explore a limited number of event sequences of a specific length in a given time. These tools can also unknowingly re-explore events/sequences leading to the same application behavior and give no indication of what percentage of the application's behavior was explored. Additionally, dynamic and static analysis tools require modeling of unavailable dependencies in order to explore certain application behavior.

In this work we investigated how model checking can improve the effectiveness (measured in code coverage) and efficiency (number and length of event sequences) when compared to current dynamic analysis

Stellenbosch University https://scholar.sun.ac.za

tools for Android applications. These tools currently obtain low code coverage for applications that depend on many external components. Model checking supports non-deterministic choices in application execution that allows systematic exploration of the application paths. Android applications include non-deterministic choices as a result of different thread interleavings, alternative event sequences and different environment configurations. Model checking also provides state matching and backtracking functionality that reduces the number of explored states and the length of execution paths of the application. Lastly, model checking provides verification of properties specified as listeners that can track and analyze the execution of the application during runtime.

We implemented our approach as an extension to Java PathFinder (JPF), called JPF-Android, to verify Android applications. JPF is a program model checker and analysis tool for Java applications and Android applications are written in Java. In order to run Android applications on the tool, outside of their original environment in an emulator, an extensive environment model is required to execute applications. The environment model consists of two components: dependency modeling and driver (event) generation. Modeling dependencies is a complex and error-prone task that requires in-depth knowledge of the application and its environment. Dependencies are usually part of a larger component and work together to perform important tasks. As a result they have complex class hierarchies and may depend on other classes and components to set up the component. We investigated tools to generate dependency models automatically. A lot of work has been done on the topic and we found that dependencies typically require a combination of techniques to be modeled. To improve code coverage we extended OCSEGen, an environment generation tool, to use runtime collected environment data for model generation. In certain cases more involved models were required and therefore classes were modeled manually. We used the tools to generate skeletons of the classes retaining side-effects to reduce manual work.

A driver is required to generate and process events to execute the entry-points of the applications. JPF-Android's driver is built from the

Stellenbosch University https://scholar.sun.ac.za

original Android application framework code using a message-queue and a single main thread to serialize incoming events for the application. When the queue is empty, a set of events is retrieved from the new JPF-Android event producer and processed non-deterministically. The event producer collects events by exploring enabled entry-points and filtering events using different event generation strategies. The tool implements three strategies: script, default and heuristic. The script event generator allows the user to write scripts containing non-deterministic event sequences. Scripts are useful to explore hard-to-reach application behavior but are tedious to write. For this reason we implemented the default event generator to automatically explore all event sequences systematically. This generator is especially useful to explore the behavior of an unknown application. For large applications, however, the number and length of event sequences are too many and too long. The heuristic event generator limits the number of times an event can be processed in a branch to reduce analysis size. Although not all paths of the application are explored in this case, the approach achieves code coverage comparable to the default event generator while exploring fewer event sequences.

Adjustments, optimizations and extensions were made to JPF to effectively apply non-determinism and state matching to Android applications. Firstly, we add functionality to configure how each choice type (thread, environment or event choices) is explored. The different choice types can be configured independently to be explored non-deterministically, heuristically or randomly. This functionality allows us to reduce the analysis size for applications with too many environment data choices by only exploring a choice once and then reusing the value. We also use this functionality to only explore a single thread interleaving that always processes the last started thread before continuing back to the previous thread. We use this as our default thread policy since combining thread choices with environment and event choices results in a too big state space to process for applications with many threads.

We also found the current approach to state matching of JPF too conservative, resulting in many unnecessary states being explored. To optimize

Stellenbosch University https://scholar.sun.ac.za

state matching, we preload application and environment classes and exclude classes from the state with none or only final static fields to improve state matching. We also create an extension to JPF called State Comparator to detect constantly changing fields and to remove them from the state to ensure a closed, finite state environment.

We implemented listeners to track all explored event sequences and environment configurations. They are used to present statistics on the explored sequences and print out the event sequence leading to an error. Erroneous sequences can be executed on the emulator, assuming it is possible to set the specific device configuration (not always supported by the emulator/device).

Android application have a multi-threaded, event-driven and asynchronous message-passing design which makes it hard to specify properties using only LTL formulas. To simplify property specification, we implement a framework to verify the execution of the application using Checklists. Checklists consist of a sequence of Checkpoints that represent methods invocations by specific threads. Checklists are verified per thread and limited to the execution triggered by processing a single event. A copy of the current Checklists is passed to newly spawned threads as well as sent with messages passed between threads to follow execution.

We evaluated the tool by comparing it to two state-of-art dynamic analysis tools for Android applications: Dynodroid and Sapienz. We found that although Dynodroid and Sapienz require less manual effort to set up, JPF-Android's flexibility allows it to cover code that is hard-to-reach with traditional dynamic analysis tools. For most applications we achieve higher code coverage using much shorter sequences. We then compared different tool configurations and found that our optimizations to JPF reduced the number of explored states while only having a small influence on the code coverage achieved by the tool.

Stellenbosch University https://scholar.sun.ac.za

7.2 Limitations and Future Work

JPF-Android is designed to support the testing of only a single Android application. This allows the simplification of the models of the Android software stack. JPF supports multiple processes and can be extended to support multiple applications. Since Android applications have a component-based design, they frequently make use of components of other applications to perform certain tasks. The biggest challenge with this extension will be to extend the system services and Binder Interprocess Communication (IPC).

Currently, we can run a discovered event sequence on the emulator or device if the environment can be configured accordingly. A next step would be to extend the currently available behavior of the emulator to allow better configuration of its environment and generate JUnit test suites from event sequences discovered on JPF-Android to run on the emulator.

Model checking suffers from the state explosion problem. We can reduce the number of possible states by using partial-order-reduction or dependency analysis to reduce the number of explored event and environment choice combinations before they are explored. Although JPF implements partial-order-reduction, this is hindered by the fact that events are generated at runtime.

This work did not focus on bug detection or detecting Android-specific errors such as resource leaks but on investigating how modeling checking can be applied effectively to Android applications. Future work can look into bug-seeding to evaluate the effectiveness of the tool.

In terms of environment generation 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 generation and side-effect analysis instead of inspecting them manually. Another improvement is to extend and refine the slicing capabilities of Modgen to

Stellenbosch University https://scholar.sun.ac.za

generate more efficient models. Symbolic execution can also be used as a complementary approach to identify entry-point parameters or return values for models to improve coverage.

Currently, dependency models can be stubbed too much, contain errors or could be modeled incorrectly. This can result in missed bugs or false positives being reported by the tool. In future work, models can be verified by comparing method traces of an application on the emulator to traces generated by JPF-Android. This might not be so simple, however, since the emulator does not have control over all environment dependencies.

New event generation strategies can be implemented, for example, to explore exceptional code.

Checklist can be recorded automatically for a specific application release and then be reused to verify the application's correctness in future releases. They can also be detected while running the application on the emulator and verify the behavior of the application on JPF-Android.

7.3 Summary

This work has shown that model checking can be effectively applied to Android applications. This approach, however, requires a lot of work to implement and presents many challenges due to Android's design. Android applications depend heavily on their environment. The environment, consisting of interdependent components, can be configured in many different ways and can be changed dynamically during runtime. It is clear that only a subset of all possible configuration combinations and event sequences are required to exercise application behavior, but, the challenge lies in choosing which combinations to explore.

Evaluation of the tool showed that our model checking approach achieved higher code coverage and explored fewer event sequences than two state-of-the-art dynamic analysis tools. It also showed that state matching and our heuristic search was effective in reducing the analysis size while not influencing the code coverage of the analysis. In our experi-

Stellenbosch University https://scholar.sun.ac.za

ence, however, we found that directly comparing different approaches, such as search-based, dynamic or symbolic execution, can be imprecise; each tool executes differently, support different application functionality and has to cut its own corners in order to run the applications. Some of these corners are described in the related papers, but some are not discussed, making it hard for researchers new to the field to interpret and improve on current results.

Stellenbosch University https://scholar.sun.ac.za

Authored and Co-authored Publications

Heila Botha, Brink van der Merwe, Willem Visser, and Oksana Tkachuk. Addressing Challenges In Obtaining High Coverage When Model Checking Android Applications. In *Proceedings of the 2017 International Symposium on Model Checking of Software*, SPIN'17, pp. 31–40, July 2017.

Guangdong Bai, Quanqi Ye, Yongzheng Wu, Heila van der Merwe, Jun Sun, Yang Liu, Jin Song Dong, Willem Visser. Towards Model Checking Android Applications. *IEEE Transactions on Software Engineering*. 2017.

Heila Botha, Brink van der Merwe, Willem Visser, and Oksana Tkachuk. StateComparator: Detecting Unbounded Variables Using JPF. *SIGSOFT Software Engineering Notes 41(6):1-5,* January 2017.

Heila van der Merwe, Oksana Tkachuk, Sean Nel, Brink van der Merwe, and Willem Visser. Environment Modeling Using Runtime Values for JPF-Android. *SIGSOFT Software Engineering Notes 40(6):1-5*, November 2015.

Stellenbosch University https://scholar.sun.ac.za

Heila van der Merwe. Verification of Android Applications. In *Proceedings of the 37th International Conference on Software Engineering* IEEE Press, Piscataway, NJ, USA, pp. 931-934, 2015.

Heila van der Merwe, Oksana Tkachuk, Brink van der Merwe, and Willem Visser. Generation of Library Models for Verification of Android Applications. *SIGSOFT Software Engineering Notes 40(1):1-5,* February 2015

Heila van der Merwe, Brink van der Merwe, and Willem Visser. Execution and property specifications for JPF-android. *SIGSOFT Software Engineering Notes 39(1):1-5,* February 2014.

Heila van der Merwe, Brink van der Merwe, and Willem Visser. Verifying Android Applications using Java PathFinder. *SIGSOFT Software Engineering Notes 37(6):1-5,* November 2012.

Stellenbosch University https://scholar.sun.ac.za

List of References

[1] Cuong, NA and Cheng, KS (2001). Towards Automation of LTL Verification for Java Pathfinder. (online). Available at: http://www.nus.edu.sg/nurop/2010/Proceedings/SoC/NUROP_Congress_Nguyen_Anh_Cuong.pdf [Accessed: 2017-05-17].

[2] EMMA Website (2006). EMMA: A Free Java Code CoverageTool. (online). Available at: http://emma.sourceforge.net [Accessed: 2017-05-17].

[3] Androguard Repository (2011). Androguard: Reverse engineering, malware and goodware analysis of android applications... and more (ninja!). (online). Available at: https://github.com/androguard/androguard [Accessed: 2017-05-17].

[4] Google Play Store (2013). Auto Answer Incoming Call. (online). Available at: https://play.google.com/store/apps/details?id=com.mathalogic.autoanswer [Accessed: 2017-05-17].

[5] Robotium Repository (2016). Robotium User Scenario Testing for Android. (online). Available at: https://github.com/robotiumtech/robotium [Accessed: 2017-05-17].

[6] Docker Website (2017). A Better Way to Build Apps. (online). Available at: https://www.docker.com [Accessed: 2017-05-17].

[7] Appium Website (2017). Appium - Automation for Apps. (online). Available at: http://appium.io [Accessed: 2017-05-17].

[8] Calabash Website (2017). Calabash - Automated acceptance testing for mobile apps. (online). Available at: http://calaba.sh [Accessed: 2017-05-17].

Stellenbosch University https://scholar.sun.ac.za

[9] Android Developer Documentation (2017). Espresso. (online). Available at: `https://developer.android.com/training/testing/espresso/index.html` [Accessed: 2017-05-17].

[10] Bitbucket Repository (2017). FlowLogger Repo. (online). Available at: `https://bitbucket.org/heila/flowlogger` [Accessed: 2017-05-17].

[11] Bitbucket Repository (2017). jpf-android-docker. (online). Available at: `https://bitbucket.org/heila/jpf-android-docker` [Accessed: 2017-05-17].

[12] Java PathFinder website (2017). jpf-core mecurial repository. (online). Available at: `http://babelfish.arc.nasa.gov/hg/jpf/jpf-core` [Accessed: 2017-05-17].

[13] Mockito Website (2017). Mockito: Tasty mocking framework for unit tests in Java. (online). Available at: `https://github.com/mockito/mockito` [Accessed: 2017-05-17].

[14] Mongo Website (2017). MongoDb Homepage. (online). Available at: `https://www.mongodb.com/` [Accessed: 2017-05-17].

[15] Android Developer Documentation (2017). Processes and Threads. (online). Available at: `https://developer.android.com/guide/components/processes-and-threads.html` [Accessed: 2017-05-17].

[16] Robolectric (2017). Robolectric: Test Drive Your Android Code. (online). Available at: `http://robolectric.org/` [Accessed: 2017-05-17].

[17] Android Developer Documentation (2017). Test Your App. (online). Available at: `https://developer.android.com/studio/test` [Accessed: 2017-05-17].

[18] Android Open Source Project Documentation (2017). The Android Source Code. (online). Available at: `http://source.android.com` [Accessed: 2017-05-17].

Stellenbosch University https://scholar.sun.ac.za

[19] Android Developer Documentation (2017). UI Automator. (online). Available at: `https://developer.android.com/training/testing/ui-automator.html` [Accessed: 2017-05-17].

[20] Android Developer Documentation (2017). UI/Application Exerciser Monkey. (online). Available at: `https://developer.android.com/studio/test/monkey.html` [Accessed: 2017-05-17].

[21] XStream Website (2017). Xstream Homepage. (online). Available at: `http://x-stream.github.io` [Accessed: 2017-05-17].

[22] Anand, S., Naik, M., Harrold, M. J. and Yang, H. (2012). *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. Automated Concolic Testing of Smartphone Apps. Cary, North Carolina. ACM New York, NY, USA.

[23] Arzt, S. and Bodden, E. (2016). *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. StubDroid: Automatic Inference of Precise Data-Flow Summaries for the Android Framework. Austin, Texas. ACM New York, NY, USA, pp. 725–735.

[24] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D. and McDaniel, P. (2014). FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. *ACM SIGPLAN Notices*, **49**(6), pp. 259–269.

[25] Barlas, E. and Bultan, T. (2007). *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering (ASE '07)*. Netstub: A Framework for Verification of Distributed Java Applications. Atlanta, Georgia, USA. ACM New York, NY, USA, pp. 24–33.

[26] Ceccarello, M. and Tkachuk, O. (2014). Automated Generation of Model Classes for Java PathFinder. *ACM SIGSOFT Software Engineering Notes*, **39**(1), pp. 1–5.

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

Stellenbosch University https://scholar.sun.ac.za

[28] Choudhary, S. R., Gorla, A. and Orso, A. (2015). *30th IEEE/ACM International Conference on Automated Software Engineering*. Automated Test Input Generation for Android: Are We There Yet? Lincoln, NE, USA. IEEE, pp. 429–440.

[29] Clarke, E. M. (2008). Grumberg, O. and Veith, H., eds., *25 Years of Model Checking*. The Birth of Model Checking, Springer Berlin Heidelberg, Lecture Notes in Computer Science, pp. 1–26. ISBN 978-3-540-69849-4. doi:10.1007/978-3-540-69850-0_1.

[30] Dwyer, M., Robby, Tkachuk, O. and Visser, W. (2004). *Proceedings of the 19th International Conference on Automated Software Engineering*. Analyzing Interaction Orderings with Model Checking. Linz, Austria. IEEE, pp. 154–163.

[31] Ehringer, D. (2010). The Dalvik Virtual Machine Architecture. (online). Available at: `http://davidehringer.com/software/android/The_Dalvik_Virtual_Machine.pdf`.

[32] Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., Sheth, A. N., Han, S., Tendulkar, V., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P. and Sheth, A. N. (2014). TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, **32**(2), p. 5.

[33] Fuchs, A. P., Chaudhuri, A. and Foster, J. S. (2009). SCanDroid : Automated Security Certification of Android Applications. (online). Available at: `https://www.cs.umd.edu/~avik/papers/scandroidascaa.pdf`.

[34] Ganov, S., Killmar, C., Khurshid, S. and Perry, D. E. (2009). *Proceedings of the 11th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering*. Event Listener Analysis and Symbolic Execution for Testing GUI Applications. Berlin, Heidelberg. Springer-Verlag, ICFEM '09, pp. 69–87.

[35] Ganov, S. R., Killmar, C., Khurshid, S. and Perry, D. E. (2008). *Proceedings of the 3rd International Workshop on Automation of Software Test*. Test generation for graphical user interfaces based on symbolic execution. New York, NY, USA. ACM, AST '08, pp. 33–40.

Stellenbosch University https://scholar.sun.ac.za

[36] Google Developer Documentation (2013). MonkeyRunner. Google. (online). Available at: `http://developer.android.com/tools/help/monkeyrunner_concepts.html` [Accessed: 2017-05-17].

[37] Goucher, A. and Riley, T. (2009). *Beautiful Testing: Leading Professionals Reveal How They Improve Software*. O'Reilly Media, Inc.

[38] Hatcliff, J., Dwyer, M. B., Păsăreanu, C. S. and Robby (2002). Foundations of the Bandera Abstraction Tools. *The Essence of Computation: Complexity, Analysis, Transformation*, pp. 172–203.

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

[40] Holzmann, G. (1997). The model checker SPIN. *IEEE Transactions on Software Engineering*, **23**(5), pp. 279–295.

[41] Hu, G., Yuan, X., Tang, Y. and Yang, J. (2014). *Proceedings of the Ninth European Conference on Computer Systems*. Efficiently, effectively detecting mobile app bugs with AppDoctor. Amsterdam, The Netherlands. ACM ACM New York, NY, USA, pp. 1–15.

[42] (2012). Symdroid: Symbolic execution for dalvik bytecode. Jeon, J., Micinski, K. K. and Foster, J. S. (online). Available at: `www.cs.umd.edu/~jfoster/papers/cs-tr-5022.pdf` [Accessed: 2017-05-17].

[43] Jeon, J., Qiu, X., Fetter-Degges, J., Foster, J. S. and Solar-Lezama, A. (2016). *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. Synthesizing framework models for symbolic execution. Austin, Texas. ACM New York, NY, USA, pp. 156–167.

[44] Jhala, R. and Majumdar, R. (2009). Software model checking. *ACM Computing Surveys (CSUR)*, **41**(4), pp. 1–54.

[45] Kim, K., Yavuz-Kahveci, T. and Sanders, B. A. (2009). *2009 IEEE/ACM International Conference on Automated Software Engineering*. Precise data race detection in a relaxed memory model using heuristic-based model checking. pp. 495–499.

Stellenbosch University https://scholar.sun.ac.za

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

[47] Lam, P., Bodden, E., Lhoták, O. and Hendren, L. (2001). The Soot framework for Java program analysis: a retrospective. (online). Available at: `https://sable.github.io/soot`.

[48] Lerda, F. and Visser, W. (2001). *Proceedings of the 8th International SPIN Workshop on Model Checking of Software*. Addressing Dynamic Issues of Program Model Checking. Toronto, Ontario, Canada. Springer-Verlag New York, pp. 80–102.

[49] Leungwattanakit, W., Artho, C., Hagiya, M., Tanabe, Y., Yamamoto, M. and Takahashi, K. (2014). Modular Software Model Checking for Distributed Systems. *IEEE Transactions on Software Engineering*, **40**(5), pp. 483–501.

[50] Li, L., Bartel, A., Bissyande, T. F., Klein, J., Le Traon, Y., Arzt, S., Rasthofer, S., Bodden, E., Octeau, D. and McDaniel, P. (2015). *IEEE/ACM 37th IEEE International Conference on Software Engineering*. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. Florence, Italy. IEEE, vol. 1, pp. 280–291.

[51] Lu, L., Li, Z., Wu, Z., Lee, W. and Jiang, G. (2012). *Proceedings of the 2012 ACM conference on Computer and communications security (CCS '12)*. CHEX. Raleigh, North Carolina, USA. ACM New York, NY, USA, p. 229.

[52] Machiry, A., Tahiliani, R. and Naik, M. (2013). *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. Dynodroid: An Input Generation System for Android Apps. Saint Petersburg, Russia. ACM New York, NY, USA, ESEC/FSE 2013, p. 224.

[53] Mahmood, R., Mirzaei, N. and Malek, S. (2014). *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Evodroid: Segmented evolutionary testing of android apps. Hong Kong, China. ACM New York, NY, USA, pp. 599–609.

[54] Maiya, P., Kanade, A. and Majumdar, R. (2013). *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*.

Stellenbosch University https://scholar.sun.ac.za

Race detection for Android applications. Edinburgh, United Kingdom. ACM New York, NY, USA, vol. 49, pp. 316–325.

[55] Mao, K., Harman, M. and Jia, Y. (2016). *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA '16)*. Sapienz: multi-objective automated testing for Android applications. Saarbrücken, Germany. ACM New York, NY, USA, pp. 94–105.

[56] Mehlitz, P., Tkachuk, O. and Ujma, M. (2011). *26th IEEE/ACM International Conference on Automated Software Engineering (ASE '11)*. JPF-AWT: Model checking GUI applications. Lawrence, KS, USA. IEEE, pp. 584–587.

[57] Mirzaei, N., Garcia, J., Bagheri, H., Sadeghi, A. and Malek, S. (2016). *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. Reducing Combinatorics in GUI Testing of Android Applications. Austin, Texas. ACM New York, NY, USA, pp. 559–570.

[58] Păsăreanu, C. S., Visser, W., Bushnell, D., Geldenhuys, J., Mehlitz, P. and Rungta, N. (2013). Symbolic PathFinder: Integrating Symbolic Execution with Model Checking for Java Bytecode Analysis. *Automated Software Engineering*, **20**(3), pp. 391–425.

[59] Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S. and Song, D. (2010). *Proceedings of the 2010 IEEE Symposium on Security and Privacy*. A symbolic execution framework for javascript. Washington, DC, USA. IEEE Computer Society, SP '10, pp. 513–528.

[60] Shafiei, N. and Mehlitz, P. (2014). Extending JPF to Verify Distributed Systems. *ACM SIGSOFT Software Engineering Notes*, **39**(1), pp. 1–5.

[61] Tkachuk, O. (2013). *Proceedings of the 2nd International Workshop on State Of the Art in Java Program analysis*. OCSEGen: Open components and systems environment generator. Seattle, Washington. ACM Press, pp. 2–5.

[62] van der Merwe, H., Tkachuk, O., Nel, S., van der Merwe, B. and Visser, W. (2015). Environment Modeling Using Runtime Values for JPF-Android. *ACM SIGSOFT Software Engineering Notes*, **40**(6), pp. 1–5.

Stellenbosch University https://scholar.sun.ac.za

[63] Visser, W., Havelund, K., Brat, G., Park, S. and Lerda, F. (2003). Model Checking Programs. *International Journal of Automated Software Engineering*, **10**(2), pp. 203 – 232.

[64] Wu, T., Liu, J., Xu, Z., Guo, C., Zhang, Y., Yan, J. and Zhang, J. (2016). Light-Weight, Inter-Procedural and Callback-Aware Resource Leak Detection for Android Apps. *IEEE Transactions on Software Engineering*, **42**(11), pp. 1054–1076.

[65] Yang, W., Prasad, M. R. and Xie, T. (2013). *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering (FASE'13)*. A Grey-Box Approach for Automated GUI-Model Generation of Mobile Applications. Rome, Italy. Springer Berlin Heidelberg, vol. 7793 of *Lecture Notes in Computer Science*, pp. 250–265.

[66] Yang, Z., Yang, M., Zhang, Y., Gu, G., Ning, P. and Wang, X. S. (2013). *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*. Appintent: analyzing sensitive data transmission in android for privacy leakage detection. New York, NY, USA. ACM, CCS '13, pp. 1043–1054.

[67] Yepang Liu, Chang Xu, Cheung, S. C. and Jian Lu (2014). GreenDroid: Automated Diagnosis of Energy Inefficiency for Smartphone Applications. *IEEE Transactions on Software Engineering*, **40**(9), pp. 911–940.