# Addressing Challenges In Obtaining High Coverage When Model Checking Android Applications

Heila Botha
CAIR, Meraka, CSIR
Department of Computer Science,
University of Stellenbosch
Stellenbosch, South Africa
hvdmerwe@cs.sun.ac.za

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

Brink van der Merwe
Willem Visser
Department of Computer Science,
University of Stellenbosch
Stellenbosch, South Africa
abvdm@sun.ac.za
visserw@sun.ac.za

## ABSTRACT

Current dynamic analysis tools for Android applications do not get good 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 we use model checking to systematically explore application paths while reducing the analysis size 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 to make this a reality 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.

## CCS CONCEPTS

**•Software and its engineering → Formal software verification;**

## KEYWORDS

Model Checking, Java PathFinder, Android applications, Dynamic Analysis

**ACM Reference format:**
Heila Botha, Oksana Tkachuk, Brink van der Merwe, and Willem Visser. 2017. Addressing Challenges In Obtaining High Coverage When Model Checking Android Applications. In *Proceedings of International Symposium on Model Checking of Software, Santa Barbara, California USA, July 2017 (SPIN'17),* 10 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

Android applications (apps) are used for banking, shopping and accessing/storing personal information. These apps operate in a safety critical environment where errors/bugs can have serious effects. Android apps, like all software applications contain bugs and errors, but they are harder to dynamically analyze because of their dependency on the complex *environment* on an Android device. This environment consists of the dependencies without which an application cannot run and an event generator to drive its execution.

Most dynamic analysis tools run apps directly on an Android emulator or device [1, 16, 18–20] to avoid modeling this complex environment. They use random/heuristic event generation strategies to drive the application's execution. In order to bound the analysis they use heuristics such as sequence length or runtime. Code coverage is then used to measure the effectiveness of the tool. Although these tools obtain high coverage for certain apps, they struggle to achieve sufficient coverage for applications critically dependent on the environment state and behavior for large parts of the application code. Additionally they only cover a subset of the application's behavior and can miss critical code since there are many event combinations and environment configurations to consider. More specifically, there are two main challenges these tools face:

**Environment Configuration** Android only exposes limited functionality to configure its environment. Tools struggle to detect the configurations required by an application as well as at which point to change the configuration to explore maximum paths in the application. Environment configurations include the battery level, network state or even the state of a remote web server or file-system.

**Event Generation** Android applications require particular events at specific points in the application's execution to enable certain application code, for example, an incoming call from an exact number. Detecting these events is hard since it might be hidden in the application's implementation. Apps can also require specific event sequences to reach certain areas in the application code.

In this paper we report on our experiences of model checking Android applications to increase the coverage obtained by dynamic analysis tools using shorter, more efficient event sequences. Model checking allows systematic exploration of events sequences and environment configurations. It reduces and bounds the analysis using state matching and backtracking to only explore application paths exploring new behavior of the application. Model checking requires an environment model to enable state matching and backtracking. Although creating such a model requires a lot of effort, it can be reused and gives us full control over the environment configuration, allowing us to to increase the coverage, and event generation, to explore shorter and more effective paths.


*SPIN'17, Santa Barbara, California USA*

DOI: 10.1145/nnnnnnn.nnnnnnn

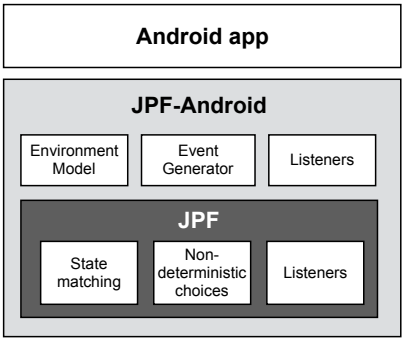


**Figure 1: Overview of JPF-Android**

We implement our approach as an extension to Java PathFinder (JPF), called JPF-Android (Figure 1). JPF is a powerful and customizable analysis engine and explicit state model checker. In order to run Android applications on JPF they must run outside of the Android emulator and provide a driver to drive the execution of the application. JPF-Android provides an *environment model* (Section 4) implementing the functionality required by all applications to run. This model is based on an abstracted version of the actual application framework and allows full control over environment configuration. The environment model also includes generated application specific models returning default/runtime/statically collected values to improve coverage. JPF-Android provides an *event generator* (Section 5) to detect enabled entry-points and generate events to drive the execution of the application. Lastly JPF-Android includes a set of listeners to record the coverage, event sequences and environment configurations explored during the analysis (Section 6.) To evaluate the effectiveness and efficiency of our approach we run the tool on a set of representative apps and show a significant increase in the code coverage using shorter event sequences in comparison to Dynodroid [16] and Sapienz [19], the current state-of-the-art dynamic analysis tools (Section 7.)

The contributions of this work include:

- Identification of the challenges of model checking Android applications,
- Implementation of solutions in JPF-Android,
- Evaluation of the effectiveness of these techniques and a comparison to state-of-the-art dynamic analysis tools.

# 2 BACKGROUND

## 2.1 Android Applications

An Android application consist of a collection of loosely coupled components. It runs on top of the extensive Android application framework, on its own Java virtual machine, in its own process. The framework provides the basic implementation of an application. It also exposes high level interfaces enabling components to interact with each other and providing access to services and libraries. The framework provides four main components that can be overwritten to implement a basic application. An *Activity* is used to control a Graphical User Interface (GUI), a *Service* performs background tasks, a *Broadcast Receiver* (BR) is used to subscribe to specific events and lastly *Content Providers* (CP) manage data access. These application components follow a specific life-cycle. The life-cycle defines the states of a component and the callbacks the framework uses to transition components between states. The application is driven by the framework in response to *events*. These event can be triggered by the user, called *User Events*, and by other applications or services generating *System Events*. Android application components are not thread safe. The stream of events sent to an application is serialized by putting them into a message queue and processing them one-by-one by the main application thread.

Running Android applications outside of the emulator has many challenges. Native libraries and services are not available, the file-system structure differs and external services are not available. This has the effect that the XML pull parsers in the framework are broken (used for GUI inflation, resource parsing, preferences) since they are implemented natively to parse compiled XML resources. Inter Process Communication (IPC) and Inter Component Communication (ICC) is also broken. This is implemented as the native Binder kernel library with hooks in the application framework to serialize communication. Local services such as SQLiteDB and the camera are also non-functional since their native drivers are not available. External operating system services responsible for application package parsing, component life-cycle management and window and input management are unavailable. These services run in the their own process, have complex implementations to support concurrent application requests and are dependent on each other and on native code.

## 2.2 Model Checking Using JPF

JPF is an open source, analysis engine for Java applications [24]. It is implemented as an explicit state model checker designed to verify Java applications at byte-code level. Explicit state model checking reduces the search space by generating application states on-the-fly, limiting the search depth and making use of state matching and backtracking to reduce the search space of an application.

In terms of model checking, the *state* of a Java application consists of the state of the heap, threads and stack. This includes the dynamic objects and their fields, loaded classes and their static variables, threads and their method traces and local variables. Java applications are not finite state by design. They depend on a (possibly) infinite 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 is not finite state. JPF allows classes to be abstracted using models to replace any Java class in the application, its libraries and even classes in the Java class library.

The application is executed serially using JPF until a *choice point* is reached in the execution where there are multiple paths forward. At this point JPF breaks the current *transition* of instructions, stores the application state and encodes the different choices in a *Choice Generator* (CG). It then systematically explores the different choices of the CG non-deterministically by branching the execution at this point and exploring each path in turn.

This state space, explored by JPF during analysis, is reduced by employing state matching and backtracking. When a new state is reached, it is compared to all the previously recorded states. If it

matches, the rest of the path has already been explored or will be explored later (depending on the search algorithm). In either case, the search is stopped for this path and backtracked to a previous choice point in the state-transition graph. Backtracking enables JPF to continue from a previous choice point without having to re-execute the path leading to the state since the state can be restored.

In this context, explicit state model checking of Java applications provides a feasible way to verify that the application code, executed by a driver, in a specific environment, is free from property violations to a certain search depth.

## 3 MOTIVATING EXAMPLE

In this section we discuss the challenges of dynamically analyzing Android applications using an app from the play store: *AutoAnswer* [1]. 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 BroadcastReceiver (BR) (Shown in Listing 1) for `PHONE_STATE` events. If the phone state changes to `STATE_RINGING` and the service is enabled in the preferences of the application (line 11-12), the contact restrictions are checked (line 15-22). If the call should be answered, the `AutoAnswerIntentService` is started to answer the call (line 25). The two main challenges for dynamic analysis tools causing low statement coverage for this application is: *environment configuration* and *event generation*.

### 3.1 Environment Configuration

Dynamic analysis tools have limited control over the environment configuration of the application which often leads to them missing important behavior because a specific configuration could not be set or there are too many configurations to consider.

This 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 the state of these components 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 the emulator this is not even possible since Bluetooth support is not emulated.

Return values from dependencies in the environment also influence AutoAnswer's coverage. The contacts CP, for example, needs to be setup with starred and not-starred contacts matching the incoming call's number before running the application to enable lines 16-21 in Listing 1. Selecting representative return values for dependencies is also hard. The application only responds to two values returned from the Bluetooth service when asked for its state: headset connect or headset not connected. If the Bluetooth service returns any other value, it will not enable any new application behavior.

[1] https://play.google.com/store/apps/details?id=com.mathalogic.autoanswer

```
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: Code extract from AutoAnswerReceiver**

### 3.2 Event Generation

Another challenge for dynamic analysis tools is generating event sequences for AutoAnswer because there are many possible events and event combinations (especially when combined with different environment configurations). The events that can be fired for this app include changing each of the settings on the preference screen as well as firing the `onReceive()` method of the BR with `PHONE_STATE` events. Since dynamic tools cannot detect that an incoming call should be fired for each environment configuration, they might fire the BR multiple times for the same configuration and miss firing it for important configurations.

Secondly, the parameters with which entry-points are called, have a big influence on the application. 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 (line 9-10). In this example this parameter is very important. If the number is null, we might run into a `NullPointerException`. To enable lines 16-21 in Listing 1, the incoming number must match a contact, a starred contact and no contacts.

## 4 ENVIRONMENT MODELING

In order to model the environment of an Android application on JPF we divide it into three components: the application framework, external services and native libraries and drivers (Figure 2).

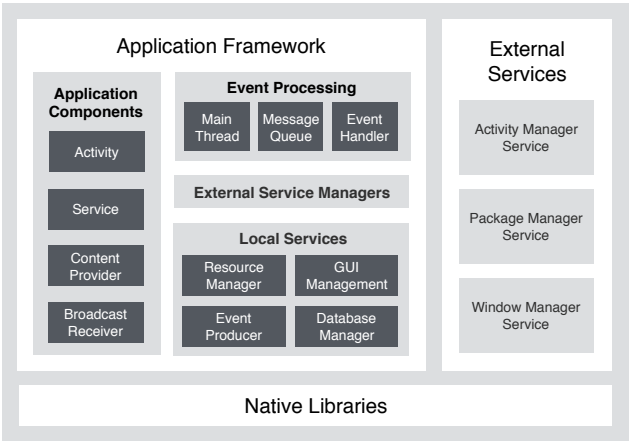


**Figure 2: The Android application environment**

The application framework is the library on which an application is built. It consists of the main application components (Activity, Service, BroadcastReceiver, Content Provider), the event processing code, local services and service managers. JPF-Android reuses the main application components from the framework since these are extended by the application. The event processing code is also reused from the framework, but additional code is added to retrieve events from the new event producer service when the message queue is empty (discussed in more detail in Section 5.) Local services often implement interfaces to native services or libraries (for example file management APIs, GUI inflation from XML, SQLiteDB, networking APIs and the camera). Depending on the usage and complexity of the services, we either abstract their implementation using manually created models (or reuse JPF's models), or generate models returning default/runtime values. Application components access external services via service managers. The three most important system services are the ActivityManager (responsible for life-cycle management of application components), PackageManager (parses the application package and returns information about the application) and the WindowManager (responsible for which window is displayed on the screen and input management). These services are greatly abstracted using manual models that run in the application process. Other services include TelephonyManager, LocationManager, NotificationManager and WiFiManager.

Models are usually created manually since this is such complex task that requires in-depth domain knowledge [4, 12, 24]. To ease to process, we use OCSEGen [22] to automatically generate models. OCSEGen is used to generate complete models for a specific set of classes or for generating models for classes referenced by an application. These models' methods return default values (false for boolean, 0 for integer and an instance of the return object if available). In previous work we extended OCSEGen to automate the process of model generation for Android apps and added functionality to generate models from runtime-collected values [23]. These models are especially useful for application specific models and return valid values to enable more of the application code. Two examples where generated runtime value models work well for modeling is SharedPreferences and cursors.

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 returning runtime observed preference values non-deterministically. This allows the tool to explore application behavior systematically for all preferences.

Cursors are used by most Android applications to traverse data retrieved from a Content Provider or database. Neither JPF nor JPF-Android attempts to store the state or backtrack the content in a database. To enable the application code using cursors, we use a default cursor implementation traversing a data set containing only a single entry. We then use generated method models mapping specific parameters to runtime return values to improve the coverage. In both of these components the models are also edited manually to throw exceptions or return unobserved values to increase coverage even further.

We use the same approach for form fields (`Checkbox`, `EditText`) throughout the application. JPF-Android does not include the state of these fields as part of the application state, but rather returns a set of possible values configured by the user non-deterministically when their values are retrieved by the application code.

Side-effects required for more specific property verification regarding API's usage patterns need to be included manually or retained using OCSEGen [22].

Over the course of the last few years we created 417 models totaling around 32k Lines of Code (LOC) [2]. The original Android application framework, excluding external services, exposes 1402 classes — many of which are reused by JPF-Android. JPF-Android models can be adapted for JUnit Testing. Table 1 lists other commonly used dependencies in the Android environment and how they are modeled.

## 5 EVENT GENERATION

Android applications require events (also called messages) to drive their execution. Events can be triggered by the application code itself or by local and external services. The application can start another Activity or start/stop a Service for example. Applications also register callbacks in local or external services. These include registering BroadcastReceivers, setting listeners on UI elements or subscribing to sensor/service updates. The *event producer* in JPF-Android only fires events that originate from the user or from external and local services — all of which are not available in JPF-Android.

Events can be fired at anytime on an Android device, but to reduce the scheduling possibilities and number of events fired, the event producer is only called when the application's message queue is empty and all threads are waiting for new events to continue execution. At this point the event producer collects the set of enabled entry-points and generates a set of corresponding events to fire non-deterministically. When no more events are returned

[2]Models are available at: https://bitbucket.org/heila/jpf-android/src

**Table 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 |

by the event producer, the analysis stops exploring that path and backtracks to a previous choice point.

An event consists of a *type* mapped to the specific entry-point of the application, a set of *parameters* used as arguments for the entry-point method and the *window name* for which it is fired. Common events fired by JPF-Android include: UIEvents, KeyPressEvents, MenuItemEvents, SharedPreferenceEvents and SystemEvents. Using correct parameters for events are important since incorrect parameters can lead to unexpected crashes in the entry-point code. Service models are required to pre-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, manually, symbolically or statically collected values stored in a database.

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 given resource limitations. For this reason we developed the event producer in such a way that it can be extended to implement different event generation strategies. The tool provides the following event generation strategies:

**Script** Allow users to write scripts containing sequences of events [23]. 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. But, writing scripts requires in-depth knowledge of the application and its environment.

**Dynamic** Fires all possible events non-deterministically.

**Heuristic** Reduces the analysis size by only firing each event once in a path.

JPF-Android starts by firing one entry-point — the main Activity defined by the application. If the application finishes this Activity and the Activity stack is empty, it stops execution in that branch (the main Activity is not restarted).

# 6 MODEL CHECKING

To analyze Android applications more efficiently we make use of explicit state model checking using JPF. JPF allows us to explore choices non-deterministically and to use state matching and backtracking to bound and reduce the search space. It also exposes a listener API to track the execution of the application at byte-code level. But all of these features come at the cost of creating more complex environment models and higher resources requirements. In this section we discuss how we utilize the functionality provided by JPF to improve the coverage and efficiency for analyzing Android applications.

## 6.1 Choices

There are three type of choices in the Android environment model: *thread choices*, *event choices* and *environment data choices*.

**Thread choices** are used to explore different thread interleavings to find deadlocks and race conditions. These choices are managed by JPF.

**Event choices** are available each time the event producer needs to fire a new event from a list of possibilities.

**Environment data choices** are used to explore different return values in environment models.

Exploring choices non-deterministically allows us to explore paths in the application systematically to ensure we do not miss paths. It also allows us to explore all choices as soon as they occur instead of having to wait until the next time the choice point is reached (if it is ever reached again). This allows us to achieve better coverage in shorter sequences.

We want to explore all possible choices to maximize the coverage the tool obtains. But the number of choice combination and states increases exponentially for each additional choice. To improve the scalability of the analysis, JPF-Android can reduce choices by only making environment choices once — the first time the choice-point is reached in a branch. At this point all choices are explored non-deterministically. The next time the choice point is reached the same choice is returned and the execution is not branched. Although all choice combinations are not explored in this case, it scales the analysis so that JPF can at least analyze all choices once.

## 6.2 State Matching

State matching bounds the execution of the application by stopping the exploration of paths when they reach a previously explored state. By default the state includes all loaded classes, all heap objects, the thread state. Native models and listeners are not included in the state or backtracked. Counters and variables keeping track of any type of history stops state matching and should be excluded

from the state. This complicates the creation of models. Classes and objects that continuously keep on changing can be detected using jpf-state-comparator and can be excluded from the state using JPF's `@FilterField` annotation or other configuration options discussed in [5].

JPF-Android further optimizes state-matching by pre-loading all application classes and application specific models. It also redefines the state of loaded classes as only the state of their non-final static fields. The main reason for these optimizations is that an event can explore behavior in the code that loads new classes but that has no other side-effects on the state. In this cases we do not want to continue the search otherwise the event would have to be executed again before the state can match.

### 6.3 Analyzing The Execution

To compare JPF-Android to current dynamic analysis tools we need to measure the code coverage of the application during execution. To ensure coverage is calculated in the same way as the other dynamic tools we implemented an EMMA coverage listener. The coverage can be calculated per branch or across all branches. The listener requires the application to be instrumented with EMMA before execution. The instrumentation fields inserted into classes by EMMA are excluded from state-matching. We then provide a model for the EMMA rt.java class to natively collect the coverage and generate a report at the end of the run.

To track the explored event sequences we provide the `EventTree Wrapper` listener storing all event sequences in an event tree. An event is added to the tree when it is returned by the event producer. The children of an event are the set of events fired by the event generator after the parent event was processed. The root of the tree is the first event that started the application i.e. the event that starts the main Activity of the application. The tree keeps a pointer to the current event processed by the framework model. When an error occurs the tool traces back the sequence of events leading to an error using the current path in the tree. This event sequence can be used to create a test case to verify the error on the emulator. The listener also keeps a list of all unique events it fired (can be used for script writing) and prints out a summary of the number of sequences and a histogram of the lengths of the sequences. The event tree is kept natively and is excluded from the application state. We do however backtrack the pointer to the current event in the tree to ensure it records correct event sequences when backtracking to another choice earlier in the tree.

JPF defines a property that is violated when uncaught or specific exceptions are thrown. We enable this property and throw an InvalidEventException if an event causes an exception in the application execution.

JPF is configured to detect race conditions and deadlocks. To detect concurrency errors JPF explores all thread interleavings while ensuring the application does not reach a state where all threads are waiting or blocked or where concurrent field accesses cause a race. Exploring all possible thread paths is a resource intensive process and can often not be combined with event and environment data choices. To reduce the search space when not looking for concurrency errors we provide a configuration option to only explore a single thread interleaving. By default the tool always executes the last started thread until it is done or waiting after which it continues to the previous thread. JPF-Android can also use a random thread schedule but then the tool need to be rerun a few times.

The depth of the search space can be bound when state matching is not possible or when state matching only occurs at such a deep level that it might not be reached it in a suitable time. JPF-Android allows bounding the depth by either limiting the state depth in the state-transition graph or the event sequence length across all paths in the event tree. However efficient, this type of bounding can lead to low coverage. Ideally we want to explore a path until a state match occurs.

## 7 EVALUATION

We evaluate the effectiveness and efficiency of our approach by analyzing a set of representative applications shown in Table 2. These applications vary in terms of LOC and number of components. The table also shows the number of application specific models generated and the number of entry-points detected by JPF-Android. This indicates the size of an application's environment. SyncMyPix, K9Mail and Keepassdroid for example have many entry-points, so they have many possible event sequences. For the first experiment we compare the statement coverage, number of event sequences and the execution time of JPF-Android to two current state-of-the-art dynamic analysis tools for Android: Dynodroid [16] and Sapienz [19]. Experiment two compares different heuristics implemented by JPF-Android and the effect they have on the coverage and the number of paths and states explored.

### 7.1 Experiment 1: Code coverage

We measure the statement coverage obtained by the tools using EMMA and exclude all classes enabling code coverage calculation for Android and external libraries. This gives a more accurate representation of the application's coverage. Code coverage 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, but it shows us what code was never analyzed [9].

Dynodroid and Sapienz analyze applications on the emulator. Dynodroid is a dynamic analysis tool that uses a biased random input generation approach, firing events more often if they are relevant in more contexts. It records the coverage and events executed. For this experiment we use the virtual-box provided by Choudhary et al. [6] to execute Dynodroid. Since Dynodroid makes use of a random input generation approach we again follow the example of Choudhary et al. [6] and execute each app for ten runs of an hour long — effectively executing ten event sequences of +/-400 events. Since we can use multiple virtual machines this accumulates to 12 hours of analysis time after which the coverage of all runs are merged. Sapienz uses a systematic Pareto-optimal multi-objective search based approach. It employs random fuzzing and string seeding to maximize coverage and minimize event sequences. Sapienz makes use of multiple concurrently running emulators to generate event sequences. The coverage of generated sequences are used to improve the generation of following event sequences. We repeat the experiment as performed in the paper by Mao et al. [19] and run each application on the tool for an hour. The tool generates a

**Table 2: 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 entry-points (EP) show the size of their environment.**

| | Version | LOC | A | S | BR | CP | Models | EP |
|---|---|---|---|---|---|---|---|---|
| **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 |

test suite and coverage reports for each test sequence which we merge to obtain the final coverage.

Before running JPF-Android on the applications, application specific models are generated for dependencies and refined to obtain higher coverage. This processes takes more or less a day depending on the the user's knowledge of the application. Once setup, these models can be reused for future runs, other applications or could be adapted for JUnit Testing. To obtain maximum coverage all possible paths in the environment must be explored by JPF-Android. But, even with all the techniques and heuristic implemented by JPF-Android, it is not always possible due to the number of choices combinations. For this experiment we configure JPF-Android to do a heuristic search, limiting events to only be executed once per Activity per path. Event sequences are bounded at length 20 and search depth is limited to 1000 states. We configure the tool to explore all environment choices once non-deterministically for each event sequence, where after the value is cached for the rest of the path. We further limit the tool to a single thread scheduling that always chooses the last started thread before returning to the other threads.

Table 3 shows the coverage achieved for each application. The highest coverage achieved for each app is highlighted. JPF-Android obtained a mean coverage of 69.5% compared to Dynodroid's 56.6% and Sapienz 50.8% over the set of apps. We found that Dynodroid and Sapienz achieve similar coverage due to the fact that they both run on the emulator which has many limitation with regards to environment configuration. Sapienz runs on a newer emulator with more behavior and in some cases can achieve better coverage due to it using seeded String values. For Dynodroid these values were hard coded to expose more application behavior. JPF-Android achieved higher or similar coverage in all but two applications: PasswordMaker and aGrep. aGrep contained a 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 depended on multiple firing of the same event in JPF-Android which was not enabled in this run to limit the search space.

JPF-Android explores all sequences systematically. It completed exploration before reaching state depth (set to 1000) or maximum event sequence length (set to 20) for all applications except for Keepassdroid. On further inspection we saw that this application has 14 Activities and so sequences of length 20 are not sufficient to cover all entry points. With JPF-Android's optimizations for preferences, the length and number of events explored cannot directly be compared to the other tools but it is clear that not firing events to change the preferences helps to shorten event sequences. The total number of paths explored by JPF-Android is given in column 5. 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. Sapienz explored sequences of length between 20-500. Dynodroid explored sequences of length between 197 to 754 events per run per app with an average of 441 events per run across all apps. The problem with these tools is that there is no way to know if maximum coverage has been reached or when to kill the analysis. Additionally we can see that only a small number or possible paths explored by JPF-Android has been explored by the other tools which leads us to the conclusion that they require longer sequences.

## 7.2 Experiment 2: Heuristics

This experiment evaluates the effect of the different heuristics implemented by JPF-Android. To evaluate the different heuristics, we compare the number of states, number of paths, the code coverage and the execution time for event sequences of length four. Table 4 shows the results.

*7.2.1 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 four instead of 20. We found that for most applications the coverage is very close to that of experiment 1. This can be explained by the fact that for most apps all entry-points are fired at least once using sequences of length four, although all paths of the application was not explored. A path can fire an entry-point again given a different context further down the path.

**Table 3: Shows the statement coverage for Sapienz (S), Dynodroid (D) and JPF-Android (J). For JPF-Android we also show the number of new states explored (#S), the number of Paths (#P) number of environment choice points (#C) and 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 |

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

| | 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 |

Applications with many entry-points can require longer sequences to reach entry-points. Keepassdroid, for example, has 69 entry-points and 14 Activities and its coverage deceases by 14% when shortening the sequences to length four.

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 only fires each event once in a branch. The results show that the number of states and paths increases significantly for such short event sequences while only improving to coverage of a subset of the apps by less than 5% each.

*7.2.2 State Matching.* To bound sequence length we use state matching to stop exploring a path when it reaches a previously visited state. The column entitled "No state matching" shows the results of disabling state matching. We can see that the number of states explodes and that three of the 12 apps do not complete or hit the state depth bound of 1000 within five hours. The results of this run highlight the actual size of the state space for these applications and the reduction in the number of states when using state matching.

*7.2.3 Runtime Values.* JPF-Android uses runtime collected values in preferences, cursors and parameters for events. In the "No RV" column we show the decrease in coverage when runtime values are disabled. The coverage reduces by an average of 10%. Additionally removing runtime values and using default values made 8 of the 12 apps crash during the analysis.

## 7.3 Discussion

*7.3.1 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 faces include:

**Dead code** All the applications contains dead code. Some of it was left over from a previous version of the applications or even a

commented out menu item or dialog. Covering this code is impossible for the other tools including JPF-Android.

**Exceptions** Java applications contain many try-catch blocks. Since JPF-Android models the environment it can throw exceptions and return invalid values to cover all exceptional code. Enabling this behavior while verifying the entire application explodes the already large state space.

**File system** For certain applications such as Keepassdroid, PasswordMaker and Ringdroid the application executes different code depending on the files on the file system. JPF-Android does not backtrack file creation/deletion or state since this information requires too much resources to keep a copy of the files for each state. It does, however, allow the application to create and delete files to enable code.

**Database** Backtracking the state of a DB is not feasible when using JPF. We found that returning runtime and default values from cursors achieves acceptable coverage for many applications. But some applications use too many cursors returning different data so that runtime values had to be reduced in order the make analysis feasible.

**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 achieved a coverage of 56% compared to its 45% reported for the default JPF-Android threading policy.

**Event Generation** The heuristics we use to reduce the event sequences are not always effective enough to reach all paths in the application.

*7.3.2 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 if device configuration changes are counted as events. In JPF-Android the number of event sequences differs depending on whether we count a sequence twice if it happens for another environment configuration. That is the reason why we rather report the number of paths explored by JPF-Android.

*7.3.3 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 the 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. In future work this can be done by verifying that the paths generated for each application is possible on the emulator. This might not be so simple, however, since the emulator does not have control over all environment configurations.

*7.3.4 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 an unique crash in four of the twelve apps. In two of the applications, Ringdroid and PasswordMakerPro JPF-Android did not cover the lines of code where the exception occurred. The exception in K9mail was a crash in the framework code. 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.

## 8 RELATED WORK

There is a whole body of work using static analysis for detecting security and privacy violations in Android applications [3, 7, 8, 13, 15]. Two drawbacks of using static analysis is that it reports false positives and does not provide an event sequence or environment configuration to dynamically verify that errors exists. It also requires modeling for unavailable or native code. Work has been done to model external code for static analysis using data flow summaries [2]. Our work, however, focuses on performing path sensitive dynamic analysis.

Many of the dynamic analysis tools for Android apps run apps on the emulator to simplify testing because of the complex environment of Android applications. Monkey [3], 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 [16] is built on the Monkey API and focuses on improving coverage by using a heuristic event generation approach. Sapienz [19] 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. Other tools instrument applications and then analyze their logs to identify errors [10], resource leaks or race conditions [18]. These tools only consider a single event sequence and need to be run several times to obtain sufficient results. Tools such as Evodroid [17], Trimdroid [20] and MagiC generate Android or Robotium [4] JUnit test cases to exercise the application code. 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 state. They also need to generate event sequences and specific event parameters to enable code coverage.

To help developers improve code coverage for JUnit tests, the Android SDK provides a few mock classes used to test application behavior. There are also mocking frameworks such as mockito that generate stubs for classes. These models differ too much from JPF-Android's requirements to re-use them. For unit and component testing we test smaller parts of the application so the model environment is smaller and less complex. To enable component interaction and correct life-cycle management of components, we require a more complex model of the environment. Recent work [11] also started to use design pattern recognition of dependencies to model

[3] http://developer.android.com/tools/help/monkey.html
[4] http://code.google.com/p/robotium/

them automatically 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. However, even these tools and techniques can benefit from using runtime/static/symbolically collected values for their models.

Research has also been done on collecting event sequences of Android application using GUI models of the applications [25]. 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 system events that can be fired at any time while the application is running resulting in complex models.

Trimdroid [20] 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.

GreenDroid [14] 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 [21] require extensive modeling and can only verify very small parts of the app at a time (such as a specific entry point) since it is computationally very expensive. Also, symbolic analysis tools cannot analyze behavior dependent on complex objects or that performs complex computations. Symbolic execution can however be used as a complementary approach to identify entry-point parameters or return values for models to improve coverage.

## 9 CONCLUSION

JPF-Android addresses the challenges of dynamic analysis by making use of a configurable environment built on JPF to analyze applications. It allows systematic exploration of the application for different environment configurations and makes use of state matching and backtracking to reduce event sequence length It searches for the shortest event sequences to errors and records all discovered event sequences and environment configurations.

In future work we plan to translate these discovered sequences to JUnit test cases to be run on the emulator. We also want to look into bug-seeding to evaluate the effectiveness of the tool. Lastly we want to use a form of partial order reduction to reduce event sequences by analyzing the effect of an event on the application state statically before firing it.

## REFERENCES


[1] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM Press, 59.

[2] Steven Arzt and Eric Bodden. 2016. StubDroid: Automatic Inference of Precise Data-flow Summaries for the Android Framework. In *Proceedings of the 38th ICSE*. ACM, New York, NY, USA, 725–735.

[3] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 259–269.

[4] Elliot Barlas and Tevfik Bultan. 2007. Netstub: A Framework for Verification of Distributed Java Applications. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering (ASE '07)*. ACM, New York, NY, USA, 24–33.

[5] Heila Botha, Brink van der Merwe, Willem Visser, and Oksana Tkachuk. 2017. StateComparator: Detecting Unbounded Variables Using JPF. *SIGSOFT Softw. Eng. Notes* 41, 6 (Jan. 2017), 1–5.

[6] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet?. In *30th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 429–440.

[7] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. 2014. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM TOCS* 32, 2 (2014), 5.

[8] Adam P Fuchs, Avik Chaudhuri, and Jeffrey S Foster. 2009. Scandroid: Automated security certification of android. (2009).

[9] Adam Goucher and Tim Riley. 2009. *Beautiful Testing: Leading Professionals Reveal How They Improve Software.* " O'Reilly Media, Inc.".

[10] Cuixiong Hu and Iulian Neamtiu. 2011. Automating GUI Testing for Android Applications. In *Proceedings of the 6th International Workshop on Automation of Software Test (AST '11)*. ACM, New York, NY, USA, 77–83.

[11] Jinseong Jeon, Xiaokang Qiu, Jonathan Fetter-Degges, Jeffrey S. Foster, and Armando Solar-Lezama. 2016. Synthesizing Framework Models for Symbolic Execution. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 156–167.

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

[13] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. IccTA: Detecting inter-component privacy leaks in Android apps. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 280–291.

[14] Y. Liu, C. Xu, S. C. Cheung, and J. L. 2014. GreenDroid: Automated Diagnosis of Energy Inefficiency for Smartphone Applications. *IEEE Transactions on Software Engineering* 40, 9 (Sept 2014), 911–940.

[15] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. 2012. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 229–240.

[16] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 224–234.

[17] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. Evodroid: Segmented evolutionary testing of android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 599–609.

[18] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. 2014. Race Detection for Android Applications. *SIGPLAN* 49, 6 (2014), 316–325.

[19] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective Automated Testing for Android Applications. In *Proc. of ISSTA'16*. 94–105.

[20] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. 2016. Reducing Combinatorics in GUI Testing of Android Applications. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 559–570.

[21] Corina. Pasareanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehlitz, and Neha Rungta. 2013. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. *Automated Software Engineering* 20, 3 (2013), 391–425.

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

[23] Heila van der Merwe, Oksana Tkachuk, Sean Nel, Brink van der Merwe, and Willem Visser. 2015. Environment Modeling Using Runtime Values for JPF-Android. *SIGSOFT Softw. Eng. Notes* 40, 6 (Nov. 2015), 1–5.

[24] Willem Visser, Klaus Havelund, Guillaume Brat, SeunJoon Park, and Flavio Lerda. 2003. Model Checking Programs. In *Automated Software Engineering*, Vol. 10. IEEE, IEEE Comput. Soc, 203 – 232.

[25] Wei Yang, MukulR. Prasad, and Tao Xie. 2013. A Grey-Box Approach for Automated GUI-Model Generation of Mobile Applications. In *Fundamental Approaches to Software Engineering*, Vittorio Cortellessa and Dniel Varr (Eds.). Lecture Notes in Computer Science, Vol. 7793. Springer Berlin Heidelberg, 250–265.