Verifying Android Applications Using Java PathFinder

Yepang Liu[1], Chang Xu[2], S.C. Cheung[1]

[1]Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong
{andrewust,scc}@cse.ust.hk

[2]State Key Laboratory for Novel Software Technology
Department of Computer Science and Technology
The Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong
changxu@nju.edu.cn

Technical Report

September 17, 2012

Abstract: Providing verification support for Android programs can benefit millions of Android application developers. This technical report describes how we extend Java PathFinder (or JPF for short), a framework for verifying general Java byte-code programs, to support Android programs. We mainly address two major technical issues. First, we derive an event handler scheduling model from Android specifications, and leverage this model to guide JPF to realistically execute Android applications in its Java virtual machine. Second, we identify a critical set of Android APIs that rely on Android system level functionalities or native libraries whose implementation are platform-specific. We model these APIs using JPF's listener and native peer mechanisms such that their side effects would not be ignored during an analysis. Our preliminary results confirmed that we can enable JPF to verify Android applications and find real defects after properly addressing these two major technical issues.

Keywords: Android Applications, Program Analysis, Model Checking, Java PathFinder

Table of Contents

1. Introduction

Increasing market penetration of smartphones fosters the proliferation various mobile applications. As of June 2012, over 500,000 applications available on Google Play Store have received 20 billions of downloads [1]. Users rely on such applications for different purposes such daily task assistance [4], entertainment [2], socializing [9] or even financial management [10]. As such, the software quality of these applications is of vital importance. Developers should extensively test their applications before shipping them. However, the reality is not optimistic. Many applications suffer different kinds of defects. A notorious example is that Android SMS application intermittently sends meaningless short messages to recipients randomly picked in the users' phone book [3]. There are two major reasons why Android applications commonly suffer different defects. First, Android applications are typically developed by small teams without dedicated quality assurance. It is not realistic for developers to perform a thorough testing of their applications on different devices. In fact, many Android applications such as K-9 Mail [17], a popular email application with millions of downloads, do not even have a well-designed test suite. Second, unlike their desktop counterparts, the history of Android applications is relatively short. There are not many mature industrial-strength tools designed for assuring the quality of Android applications. Although existing testing frameworks such as Robotium [20] can help developers write automatic black-box test cases, it still requires non-trivial efforts to write test cases that can achieve high statement or path coverage [18].

For conventional programs (e.g., Java programs), there are quite a few mature, free, and easy-to-use tools or analysis frameworks to help developers diagnose defects in their programs. For example, FindBugs [11] leverages static analysis to detect various bugs (e.g., null pointer exceptions) in Java code. Java PathFinder [22] (or JPF for short) can help model check a Java program to prove the absence of a certain type of errors. However, neither FindBugs nor JPF has a good support for Android applications[1]. In this report, we aim to discuss our attempt to enable JPF to analyze Android applications.

Extending JPF to support analyzing Android applications is a difficult task. Researchers from the "Google Summer of Code" program also aim at the same goal, but have not been too successful over the past several years [16]. Specifically, there are two major challenges to address before one can use JPF to analyze an Android application. The challenges are:

- **Challenge 1.** Android applications follow an event-driven programming paradigm, which hides an application's program control flows in the canned machinery of the Android framework. Developers are only exposed with a set of loosely coupled event handlers for them to specify an

[1] We have seen some efforts to enable FindBugs to analyze Android application. A third-party plugin can be found at http://code.google.com/p/findbugs-for-android/.

[2] In order to be able to backtrack to a previous program state, JPF sacrifices the capability of executing Java libraries that rely on

application's behavior. At runtime, these event handlers are implicitly called according to Android specifications. For example, the onStart() handler of an activity component is called after its onCreate() handler, but this calling order is never explicitly specified in the program code. This causes trouble for JPF as it is designed to verify conventional Java programs where program control flows are explicitly stated.

- **Challenge 2.** Some Android APIs rely on the Android system functionalities or native libraries whose implementations are platform-specific (e.g., thread manipulation APIs and GUI related APIs). System level functionalities or native libraries are typically implemented in C or C++, and thus their code is not suitable to be executed in JPF's Java virtual machine. If we simply choose to ignore their side effects, JPF may encounter various unexpected problems when it analyzes an Android application.

To address the first challenge, we need to derive an event handler scheduling model (or EHS model for short) from the Android specifications. This model captures application generic temporal rules that specify the calling relationships between event handlers. These rules should be enforced at runtime to guide JPF to call event handlers at the appropriate time. To address the second challenge, we need to identify those Android APIs that rely on native libraries or system functionalities, and properly model their side effects so that JPF can make the right actions when those APIs are called. By addressing these two challenges, JPF would be able to help model check Android applications.

The rest of this report is organized as follows. Section 2 gives the background of Android applications, and JPF. Section 3 introduces our EHS model and discusses how to enforce this model at runtime. Section 4 discusses the modeling of some critical Android APIs. Section 5 presents preliminary experimental results showing that our extension can successfully enable JPF to verify Android applications. Finally, Section 6 concludes this report.

2. Background

In this section, we introduce the background of Android applications, and Java PathFinder.

2.1 Android Applications

Android [1] is the most popular, open-source, Linux-based smartphone platform. Android applications are written in Java programming language with special enforcements. The applications will be first compiled into Java byte codes. Then the Java Virtual Machine-compatible .class files will be converted to Davik Virtual Machine-compatible .dex files, which can be directly executed on real devices. Typically, an Android application contains the following four types of components:

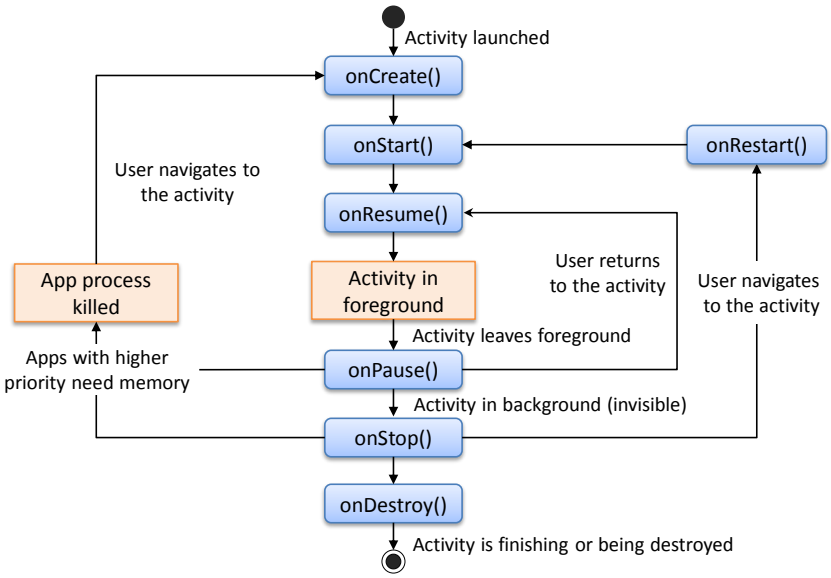

Figure 1. Activity lifecycle

- **Activity**. Activities are the only type of components that contain graphical interfaces to display information and interact with users. An application may comprise multiple activities that work together to provide a cohesive user experience. An activity can be launched from other activity or service components.
- **Service**. Services are components that run in the background for conducting long-running tasks (e.g., sensor reading or data synchronization etc.). Other components, typically activities, can start and interact with services.
- **Broadcast receiver**. Broadcast receivers define how an application responds to system-wide broadcast messages sent from other components, applications or the Android system. It can be statically registered in an application's configuration file (e.g., AndroidManifest.xml), or dynamically registered in an activity or service component at runtime.
- **Content provider**. Content providers manage shared application data persisted in file systems, databases or network locations. They provide an interface for other components or applications to query or modify these data.

Each application component has its own lifecycle defining how it is created, used, and destroyed. For example, Figure 1 shows the entire lifecycle of an activity component. An activity's lifetime starts with a call to its onCreate() handler, and ends with a call to its onDestroy() handler. An activity's foreground lifetime starts after a call to its onResume() handler, and lasts until its onPause() handler is called when another activity comes to the foreground. In the foreground, an activity can interact with its user. When it goes to the background and becomes invisible, its onStop() handler would be called. When users navigate back to a paused or stopped activity, the activity's onResume() or onRestart() handler would be called

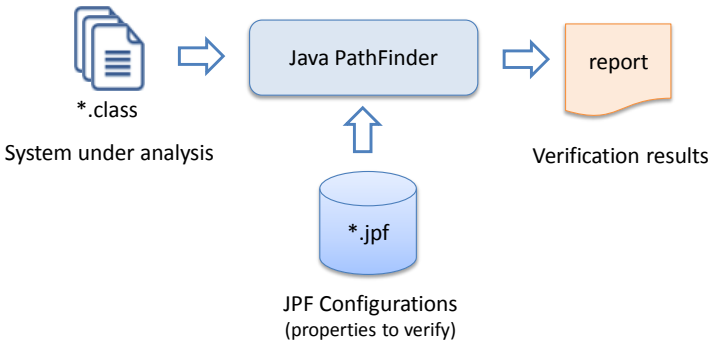

Figure 2. JPF basics

correspondingly, and the activity would then come to the foreground again. In exceptional cases, a paused or stopped activity may be killed for releasing memory to other applications with higher priorities.

2.2 Java PathFinder

Java PathFinder [15] (or JPF for short) is a highly customizable execution environment designed for verifying Java byte-code programs. As shown in Figure 2, JPF takes as input the byte codes of the system under analysis, and a set of configurations specifying the properties to check. JPF systematically analyzes the target system starting from an entry point (e.g., the main function of a system), and checks for violations of the specified properties. During the analysis process, JPF can identify the program points where the execution can proceed on different program paths. In theory, JPF would explore all such paths for verification purposes (we may also call this the model checking functionality of JPF). However, due to combinatorial explosion, the number of all paths can be unbounded, which is known as the "state explosion problem" in software model checking. JPF deals with this problem using *state matching*: when it reaches a program point where it can choose to proceed differently (e.g., an if-else branch depending on the value of a random variable), it would check whether similar program states have been explored. If yes, it would abandon the corresponding choice (i.e., stop exploring a program path), and backtrack to the previous point where it has unexplored choices[2]. By doing so, JPF model checks a target system and helps detect all defects that violate the specified properties.

JPF is highly flexible and provides many extension points, of which we are going to leverage two important ones to enable JPF to verify Android applications. They are *listeners* and *native peers*. We introduce them in Section 2.2.1 and Section 2.2.2, respectively.

2.2.1 JPF Listeners

JPF listeners provide a mechanism to monitor and interact with JPF's internal executions. Figure 3

[2] In order to be able to backtrack to a previous program state, JPF sacrifices the capability of executing Java libraries that rely on native code. This is because native code such as system calls for file writing cannot be easily reverted.

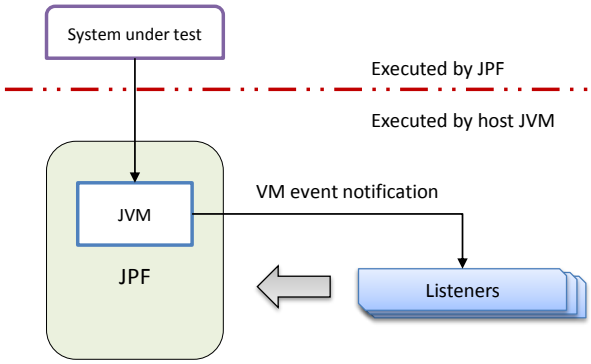

Figure 3. JPF listener mechanism

illustrates the key concept of the listener mechanism. The whole JPF framework runs inside the host JVM (i.e., the standard JVM on the host machine). At the heart of JPF is a specialized Java virtual machine for executing the byte codes of the target programs. Similar to plugins, JPF listeners can be dynamically configured at runtime and also run at the host JVM level. When there are special events occurred in JPF's JVM (e.g., a certain API is called), the configured listeners will be notified, and the actions defined in the listeners will be executed by the host JVM. As such, listeners can help monitor all execution events inside the JPF's JVM. For example, the following two methods of the ListenerAdapter class are commonly overridden.

- `public void executeInstruction(JVM vm)`
- `public void instructionExecuted(JVM vm)`

The first method will be invoked right before JPF's JVM executes a byte code instruction. The second method will be invoked right after JPF's JVM executes a byte code instruction. By overriding them, one can gain control over the execution of every single byte code instruction, and perform many analysis tasks. For instance, one can check whether an object reference returned from an "areturn" byte code instruction refers to a null object. This would help trace the causes of null pointer exceptions.

2.2.2 JPF Native Peers

Each Java virtual machine that conforms to the standard JVM specifications [21] supports interfacing certain functionalities (e.g., I/O and GUI) to the native OS level (i.e., Java Native Interface). Similarly, JPF also supports delegating the execution of some methods to the host VM level, and this mechanism is called "Model Java Interface" (or MJI for short). The key concept of MJI is illustrated in Figure 4. JPF intercepts native methods and the system level methods of some stand library classes, and delegates their execution to the host JVM. The host JVM executes these methods' native peers such that some critical side effects of these methods would not be ignored by JPF.

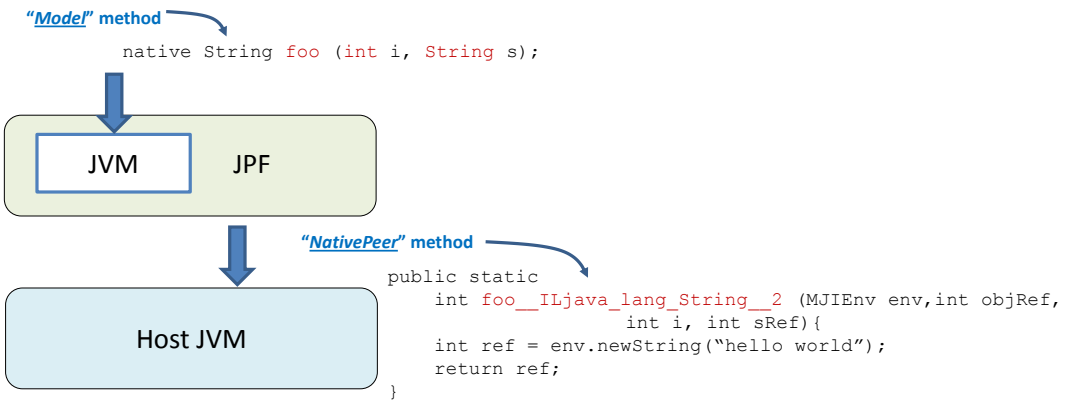

Figure 4. JPF MJI mechanism

We note that modeling the behavior of library classes (or methods) is a labor-intensive task. For example, completely modeling Android APIs that rely on native library requires enormous engineering efforts, which is typically unrealistic for individual researchers. To cope with this issue, one can choose to ignore the side effects of some methods if these effects are not relevant for the verification target. In addition to saving manual modeling efforts, this will also reduce the state space that JPF needs to explore.

3. Event Handler Scheduling Model for JPF

An Android application starts with its main activity, and ends after all its components are destroyed. It keeps handling received events by calling their handlers according to Android specifications. Each call to an event handler may change the application's state by modifying its components' local and global program data. To realistically execute Android applications in JPF's JVM, we need an event handler scheduling model (or EHS model) derived from Android specifications, and leverage this model to guide runtime scheduling of event handlers. In this section, we are going to discuss our EHS model, and how to enforce this model at runtime (i.e., when JPF analyzes an application).

3.1 EHS Model

Our EHS model is a collection of temporal rules, specifying the calling relationships between event handlers. They are generic to all Android applications and should be enforced at runtime. Formally, we define our EHS model as follows (unary temporal connective ***G*** means "always"):

$$AEM \coloneqq G \bigwedge_i R_i$$

Each temporal rule is expressed in the following form:

$$R_i \coloneqq [\psi],[\phi] \Longrightarrow \lambda$$

In a rule R_i, ψ and λ are two temporal formulae expressed in linear-time temporal logic, and refer to the past and future, respectively. ϕ is a propositional logic formula referring to the present. ψ describes

Table 1. Interpretation of temporal connectives

Temporal Connective	Type	Interpretation
G	Unary connective	Always
X	Unary connective	Next
X^{-1}	Unary connective	Previously
S_s	Binary connective	Strong since

what has happened in an execution, ϕ evaluates the current situation (what event is received), and λ describes what should be done in the future. Then the whole rule can be interpreted as:

If both ψ and ϕ hold, λ should be executed next.

We list some representative rules in the following, and Appendix A gives the entire collection of the temporal rules[3]. Propositional connectives $\wedge$, $\Rightarrow$, and $\neg$ in these example rules follow their traditional interpretations, and temporal connectives are explained in Table 1 (including those temporal connective used in the rules in Appendix A). Unary temporal connective $\boldsymbol{X}$ means "next", and its past time analogue $\boldsymbol{X^{-1}}$ means "previously". Binary temporal connective $\boldsymbol{S_s}$ means "strong since". Specifically, a temporal formula "$F_1\ \boldsymbol{S_s}\ F_2$" means that F_2 held at some time in the past, and since then F_1 always holds.

- Temporal Rule 1: When to call the lifecycle event handler act.onStart()?
  - $[\boldsymbol{X^{-1}}\ act.onCreate()],\ [\neg ACT\_FINISH\_EVENT] \Rightarrow \boldsymbol{X}\ act.onStart()$
- Temporal Rule 2: When to call a button-click event handler listener.onClick()?
  - $\big[(\neg act.onPause()\ \boldsymbol{S_s}\ act.onResume()) \wedge$
    $(\neg btn.reg(null)\ \boldsymbol{S_s}\ btn.reg(listener))\big],\ [BTN\_CLICK\_EVENT] \Rightarrow \boldsymbol{X}\ listener.onClick()$
- Temporal Rule 3: When to call a message event handler rcv.onReceive()? (Dynamic registration)
  - $[\neg \text{rcv}.unreg()\ \boldsymbol{S_s}\ \text{rcv}.reg()],\ [MSG\_EVENT] \Rightarrow \boldsymbol{X}\ \text{rcv}.onReceive()$
- Temporal Rule 4: When to call a message event handler rcv.onReceive()? (static registration)
  - $[True],\ [MSG\_EVENT] \Rightarrow \boldsymbol{X}\ \text{rcv}.onReceive()$

The first example rule states that the onStart() handler should be called after the onCreate() handler completes as long as the concerned activity does not finish. The second rule requires a button-click event handler to be called if: (1) the button is clicked, (2) its enclosing activity is in the foreground (i.e., the activity's onPause() handler has not been called since the last call to onResume() handler), and (3) its click event listener is properly registered. The third rule disenables the call to a message event handler before its registration and after its unregistration. The last rule requires that a static message event handler should be called upon any interested broadcast message.

[3] We do not claim for the completeness of our EHS model due to the complexity of Android specifications. Its current version already suffices to helping verify quite a few real-world Android applications, and it is essentially extensive. We are on the way to improve this model.

3.2 Runtime Enforcement of EHS Model

We in this subsection discuss how to enforce the EHS model during the verification process of an Android application. Since the EHS model is for scheduling event handlers, in this subsection we also cover the design details of our main scheduler, which is the analysis entry point for JPF.

The temporal rules in our EHS model are expressed in an implementation-oriented manner. They can be directly converted to a decision procedure. The decision procedure helps the main scheduler decide which event handler to be called next according to the application's execution history and its newly received events. The events come from two sources: (1) the GUI events and the main activity's start event are generated by the main scheduler; (2) other events (e.g., the event to start a service) are monitored during the application's execution (by the listener mechanism, see Section 4). The main scheduler thus should be sensitive to an application's execution history. Particularly, in our design, the main scheduler tracks the following major information:

- **A stack of active activities.** The main scheduler maintains active activities in a stack. Each active activity is associated with a collection of GUI event listeners registered for corresponding GUI elements in this activity (see Section 0 for how we build this association).
- **A list of running services.** Running services are maintained is a list. Particularly, each service launched by the binding mechanism[4] is associated with a collection of application components that are connected with it.
- **A list of registered broadcast receivers.** The main scheduler also tracks broadcast receivers. Each registered broadcast receiver (we consider that a static broadcast receiver is always registered) is associated with a filter specifying its interested message types and permission (a message sender needs to know the permission of a receiver so that it can successfully send a message to this receiver).

The scheduler serves as the analysis entry point for JPF. Figure 5 shows the pseudo code of the main function of our scheduler. It starts the application by launching its main activity (can be learned from the AndroidManifest.xml file). After that it enters a loop. In each iteration, the scheduler picks up the activity at the top of the activity stack. If this activity is ready for user interaction, the main scheduler will "randomly[5]" generate one possible GUI event (recall that each activity is associated with a collection of GUI event listeners, and the activity GUI layout can be learned by analyzing the corresponding configuration file), and query the decision procedure whether the GUI event's handler should be called

[4] A service component can be launched via the normal starting mechanism (i.e., by calling Context.startService()) or the binding mechanism (i.e., by calling Context.bindService()). The two mechanisms lead to different life cycles. See Android service component reference [6] for details.

[5] The random number can be generated using JPF's Verify.random(int *n*) method. JPF will systematically explore all possible choices from 0 to *n*-1 if JPF's configuration parameter cg.enumerate_random is set to true.

```
Add the main activity to activity stack
Start the main activity
While(stoppingCriterion not satisfied){
  get the stack top activity act
  if(act ready for interaction){
    randomly pick one GUI event from all possible GUI events
    call corresponding GUI event handler by querying the decision procedure
  } else{
    call act's corresponding life cycle event handler by querying the decision procedure
  }
}
Finish all active activities
Finish all running services
```

Figure 5. The pseudo code of our scheduler's main function

next. If yes, it will call the handler using Java reflection. If the stack top activity is not ready for user interaction, the main scheduler will query the decision procedure to determine which life cycle handler of this activity should be called next, and then call the corresponding handler. The loop will terminate when a certain criterion is satisfied. This stopping criterion can be designed very flexibly. For example, we can stop the loop if any of the following two conditions are satisfied: (1) the activity stack becomes empty, (2) no more user interaction events are allowed to be generated[6]. After the loop stops, the main scheduler will stop all active activities and running services if any.

In addition to the main function that serves as the analysis entry point of JPF, the main scheduler should also provide methods to manage the activity stack, running service list, and the broadcast receiver list. This is because during the execution of an application, many events can be generated and need to be handled on the fly. For example, the call to the startActivity() API will stop the current foreground activity, and start a new activity. As mentioned earlier, such events will be monitored using JPF's listener mechanism. When a listener is notified that startActivity() is called, it can call the corresponding method provided by the main scheduler to manage the activities. The details will be introduced in the following section.

4. Android API Modeling and Abstraction

As mentioned earlier, some Android APIs leverage Android system level functionalities or rely on native libraries. This causes big trouble to JPF. First, typically the byte codes of these Android APIs are not available for analysis. Second, even if their byte codes are obtained (e.g., we built the Android framework to get the non-stub version of android.jar), JPF's JVM will still fail to execute them because JPF has no idea about how to handle native calls. As such, we need to properly model these APIs. We note that completely and precisely modeling all these APIs and their side effects requires enormous engineering efforts. We in this section introduce our modeling of some critical Android APIs as examples

[6] All possible user interactions are infinite, but the computational resources are finite. Thus one needs to limit the number of user interaction events that can be generated during an execution of an application under analysis.

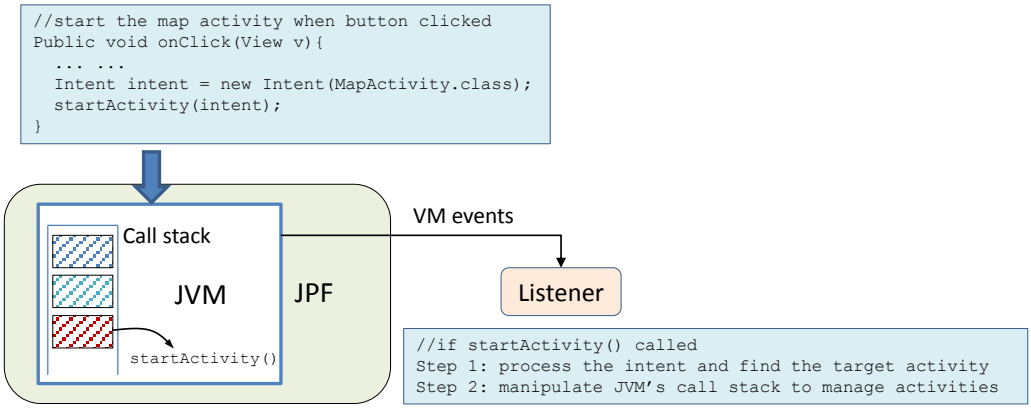

Figure 6. Modeling the startActivity() API

(the collection of APIs we have carefully modeled can be found in Appendix B). Readers can decide which APIs to model based on their analysis goals (i.e., choose to ignore the side effects of some APIs if they are not relevant to the property that is to be verified).

4.1 Activity and Service Management

Android applications (i.e., typically the activity components) can start a new activity by calling startActivity() or startActivityForResult() APIs. An active activity *act* can be finished in two ways: (1) the activity itself calls the finish() API, (2) the activity that starts *act* calls the finishActivity() API with the request code that was used to start *act*. In this subsection, we introduce the modeling of startActivity() here as an example.

The major effect of the startActivity() APIs is to stop the activity at foreground, and launch a new activity and put it in foreground. So in order to model such effects, we need to properly guide JPF to conduct the following tasks:

- **API calls interception.** We need to closely monitor an application's execution, and intercept each call to the startActivity() API.
- **Side effects abstraction.** As discussed earlier, startActivity() leverages system level functionality (e.g., thread manipulation), and relies on native code (e.g., native activity manager). As such, we need to ignore its real implementation, and abstract its side effects (i.e., we can define an empty native peer method).
- **Activity management.** After intercepting the call to startActivity() API, we need to switch from the current foreground activity to the new activity (i.e., modeling the critical side effects).

The first two tasks can be done using the listener and native peer mechanisms of JPF, respectively. The third task can be done by calling the activity management methods of our main scheduler. Figure 6 illustrates the modeling process. When the listener is notified that startActivity() is called, it will process

```
//code to manipulate the call stack of JPF's JVM
MethodInfo actSwitch = mainScheduler.getMethod("activitySwitch(Ljava/lang/Object;)V", false);
if(actSwitch != null){
  //create the call stub
  MethodInfo stub = actSwitch.createDirectCallStub("[Activity Switch]");
  DirectCallStackFrame frame = new DirectCallStackFrame(stub);
  //push the reference of the activitySwitch() method's argument onto the call stack frame
  frame.pushRef(argRef);
  vm.getLastThreadInfo().pushFrame(frame);
  vm.getLastThreadInfo().executeInstruction();
}
```

Figure 7. Code snippet to manipulate the call stack of JPF's JVM in listeners

the intent argument and find out the target activity. After that, the listener should "call" the activity management method provided by our main scheduler to perform the activity switching task.

As we have discussed in the background section, JPF and its listeners runs in the host VM, and the application under analysis runs in JPF's JVM. So if we wish to manage the activities tracked by our main scheduler (note that the main scheduler also runs in JPF's JVM), we need to manipulate the runtime call stack of JPF's JVM instead of calling the scheduler's activity management methods directly in the listener. We give the example code snippet in Figure 7. It creates a frame for the activity management method of the main scheduler, push it onto the call stack of JPF's JVM, and then starts the execution.

Similar to activities, other components can start a service by calling the startService() or bindService() APIs. A running service can also be stopped in two ways. First, a service can stop itself by calling the stopSelf() API. Second, other component can stop a running service by calling the stopService() API. Our main scheduler provides corresponding methods to manage the life cycle of service components. The modeling of these APIs are similar to the modeling of startActivity(). We do not make further elaborations here.

4.2 Broadcast Receiver Management

In Android applications, broadcast receivers can be statically registered in application configuration files or dynamically registered in the activity or service components at runtime by calling registerReceiver() API. A dynamically registered receiver can be unregistered by calling unRegisterReceiver(). Each broadcast receiver is associated with a message filter and permission (both can be optional). Other components can broadcast a message at any time by calling the sendBroadcast() API. Our main scheduler maintains a list of registered receivers (static receivers are considered as being always registered). Figure 8 illustrates how such management task is done, and how the concerned APIs are modeled using JPF's listener mechanism. As shown in the figure, the list of statically registered broadcast receivers is obtained by analyzing an application's configuration files. The dynamically registered broadcast receivers are managed by monitoring their registration and unregistration events. When a dynamic broadcast receiver is registered (i.e., the registerReceiver() API is called), our

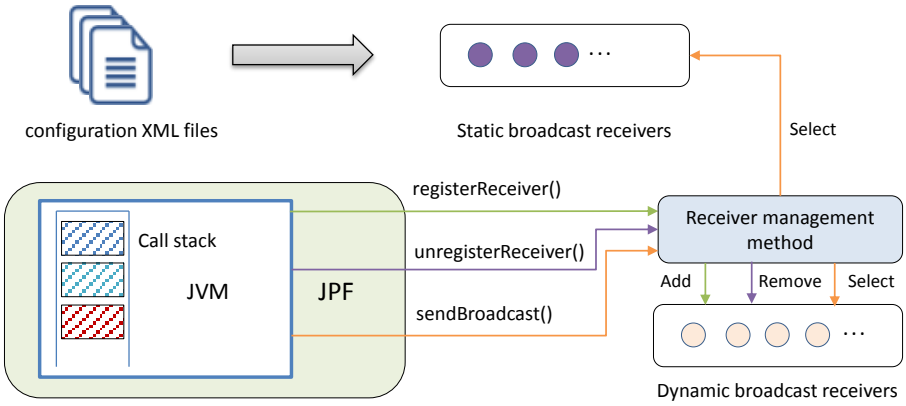

Figure 8. Broadcast receiver managment and API modeling

management method will add this receiver to the dynamic receiver list (recall that the method is called by manipulating the call stack of JPF's JVM). Similarly, when a dynamic broadcast receiver is unregistered (i.e., the unregisterReceiver() API is called), it will be removed from the dynamic receiver list. By maintaining both the static and dynamic broadcast receivers, one can then properly pick up the appropriate receiver when a message is broadcasted (i.e., the sendBroadcast() API is called) by checking the message's action and permission strings.

4.3 GUI Event Listener Management

Graphical user interfaces play a central role in Android applications. However, their construction and manipulation highly rely on native code. This makes analyzing Android applications using JPF very difficult. For example, in real executions of an Android application, when an activity is launched, the main thread (or UI thread) of the application process would construct its GUI by calling native graphical libraries (the GUI layout is learnt from the application configuration files). When the activity needs to register an event listener with a GUI element, it would first get the reference of the GUI element by calling the findViewById() API, and then perform registration as shown in the following example code.

```
Button btn = (Button) findViewById(R.id.btn);
btn.setOnClickListener(myListener);
```

We note that modeling the native graphical library is an unrealistic task. As a result, when the application is executed in JPF's JVM, the findViewById() API will not correctly return a GUI element reference. Therefore, we need to properly model the findViewById() API. Figure 9 illustrates how we model the findViewById() API. It consists of two parts: a static part and a dynamic part. In the static part, we pre-analyze an application's configuration files to learn the GUI layout of each activity component, containing the key information such as each GUI element's type and id. Then, when the application is executed by JPF, we monitor the call to the findViewById() API. When it is called, we would retrieve

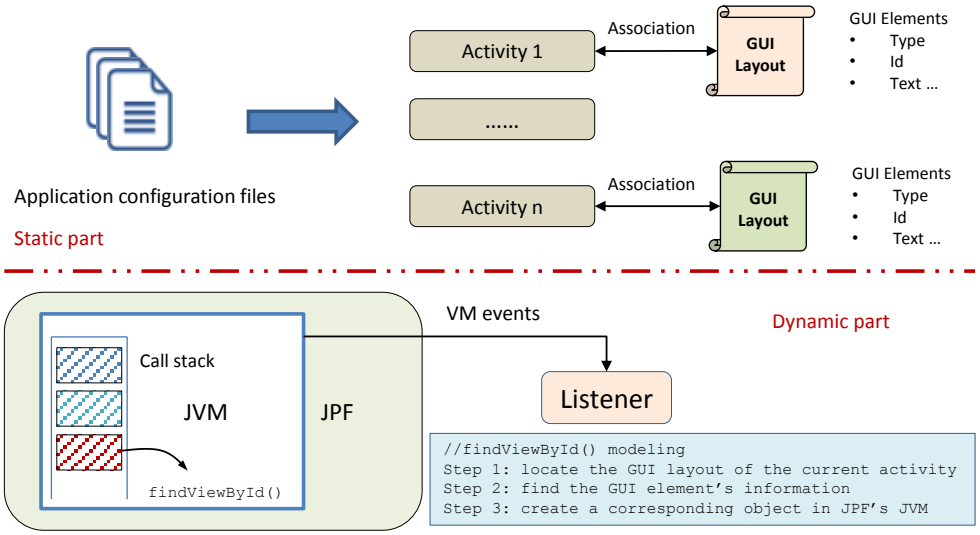

Figure 9. GUI layout analysis and the modeling of findViewById() API

from JPF's JVM the reference to the current activity, and find out the activity's layout. By doing so, we would obtain all necessary information about the corresponding GUI element. Finally, we can create an object in JPF's JVM for the GUI element, and make findViewById() return the object's reference.

After properly modeling findViewById() API, building the associations between GUI event listeners and activity components becomes straight forward. Similar to the management of dynamic broadcast receivers, we can design a listener to monitor the registration operation of a GUI event listener, and associate the event listener with the foreground activity (recall that we maintain a stack of activities). Each listener corresponds to a specific GUI event. Then we would be able to know what user interaction events can be generated when a foreground activity is ready for interaction, and leverage JPF's model checking functionality to systematically explore all possibilities.

5. Preliminaries Experimental Results

We implemented our extension on top of JPF for Android 2.3.3, which is the most popularly installed Android platform. To evaluate its usefulness, we conducted controlled experiments, which are designed to answer the following research question?

Research Question: *Can our extension enable JPF to verify real-world Android applications and find real defects? What is the analysis overhead?*

5.1 Experimental Setup and Design

We selected five open-source Android applications as our experimental subjects. They received popular downloads from Google Play Store [13] or Google Code [12]. Table 2 lists their basic

Table 2. Experimental subjects

Application	Basic Information			
	Revision No.	Lines of code	Downloads	Availability
Osmdroid	750	18,091	10,000—50,000	Google Play Store
Zmanim	322	4,893	10,000—50,000	Google Play Store
Omnidroid	863	12,427	1000—5000	Google Play Store
DroidAR	204	18,106	1000—5000	Google Code
Recycle-locator	68	3,241	1000—5000	Google Play Store

Table 3. Analysis overhead

Application	Analysis Overhead	
	Analysis Time (seconds)	Memory Consumption (MB)
Osmdroid	94	442
Zmanim	76	174
Omnidroid	163	293
DroidAR	205	188
Recycle-locator	27	115

information. For example, we choose the 322nd revision of Zmanim as one subject. It contains 4,893 lines of code, and has received more than 10,000 downloads in Google Play Store. All these applications were compiled for Android 2.3.3.

To answer our research question, we implemented a resource leak detection algorithm [7] on top of our extension. Specifically, this algorithm helps detect whether sensor listeners are properly unregistered[7] before an Android application exits. We applied this detection tool to analyze our experimental subjects. As discussed earlier, all possible user interactions are infinite. As such, we controlled our tool to generate at most six user interaction events during each application execution. Our experiments were conducted on a dual-core machine with Intel Core i5 CPU and 8GB RAM, running Windows 7 Professional SP1. We report our preliminary results in the following section.

5.2 Experimental Results

Table 3 presents our tool's analysis overhead for the five applications. Even for two largest subjects Omnidroid and DroidAR (over 18K LOC), our tool finished the verification within four minutes and cost less than 500 MB memory. Such overhead is well supported by modern PCs and compares favorably with state-of-the-art testing or debugging techniques [20].

[7] Many Android applications use sensors to provide context-aware services. To use a sensor, they need to register a listener with the Android system. When the sensors are not needed, the application needs to unregister the corresponding listener. Otherwise, it will lead to wasted sensing operations and battery energy.

Encouragingly, we found two real defects in DroidAR and Recycle Locator. DroidAR is a framework for augmented reality on Android. It leverages sensory data to digitalize the real world and make users' environment interactive. Recycle-locator is a location-aware application for helping users quickly locate services such as vending machines in university campuses. Our tool detected that their sensor listeners are never unregistered after usage [8][19]. Android system would keep an application process (even an empty process) alive as long as possible until its memory runs low [5]. As a result, such forgotten sensor unregistration could lead to wasted sensing operations, which would consume much battery energy.

5.3 Discussion

JPF is one of the most mature and robust frameworks for verifying Java byte-code programs. Since Android programs are a special type of Java programs. Extending JPF for verifying Android applications is possible. However, the extension is very difficult and requires huge engineering efforts as we have discussed in this report. Although we have successfully enabled JPF to verify quite a few Android applications, there are still many problems remaining unsolved. We discuss two major problems here.

First, we did not properly model the concurrent features of Android applications[8]. We observe that concurrency is popularly used in real-world Android applications. If the concurrent features are relevant to the verification target, ignoring concurrency will lead to imprecise and unsound analysis results.

Second, we did not properly model the dynamic GUI features of Android applications. Our extension assumes that the GUI layout of activities components would remain unchanged during an application's execution. This is true for a large proportion of Android applications. However, Android system allows an activity to change its GUI layout at runtime. To support precise and sound analysis of general Android applications, modeling such dynamic GUI features are unavoidable.

6. Conclusion

In this report, we have discussed how to extend Java PathFinder to verify Android applications. We mainly addressed two challenges. First, we derived an event handler scheduling model from Android specifications. This model can guide JPF to realistically call event handlers when analyzing Android applications. Second, we discussed how to properly model critical Android APIs that leverage Android system functionalities or rely on native libraries such that the side effects of these APIs would not be ignored by JPF. We implemented our extension for Android 2.3.3, and built a prototype tool on top of our extension to detect forgotten sensor listener unregistration defects in Android applications. Our preliminary experimental results show that the extension can help verify real-world Android applications

[8] When there is no explicit use of concurrent programming constructs, all components (including background services) of an Android application would run in the UI thread of the application process.

and locate real defects. Currently, our extension cannot apply to general Android applications because some features (e.g., concurrency and dynamic GUI) are not properly modeled. In future, we are going to address these problems and enable JPF to verify more and more Android applications. This would benefit millions of Android developers as JPF can help them automatically detect defects in their applications.

References

[1] "Android." URL: http://www.android.com/

[2] Angry Birds on Google Play Store. URL: https://play.google.com/store/apps/details?id=com.rovio. Angrybirds

[3] Android issue 9392. URL: http://code.google.com/p/android/issues/detail?can=1&q=9392

[4] Any.Do on Google Play Store. URL: https://play.google.com/store/apps/details?id=com.anydo

[5] "Android Process Lifecycle." URL: http://developer.android.com/reference/android/app/Activity.html# ProcessLifecycle.

[6] Android Service Component. URL: http://developer.android.com/reference/android/app/ Service.html

[7] M. Arnold, M. Vechev, and E. Yahav, "QVM: an efficient runtime for detecting defects in deployed systems," ACM Trans. Sotware Engineering and Methodology, vol. 21, pp. 2:1-2:35, 2011.

[8] "DroidAR issue 27." URL: http://code.google.com/p/droidar/issues/detail?id= 27

[9] Facebook on Google Play Store. URL: https://play.google.com/store/apps/details?id=com.facebook. Katana

[10] Financial Calculator on Google Play Store. URL: https://play.google.com/store/apps/details?id=com.financial. calculator

[11] FindBugs. URL: http://findbugs.sourceforge.net/

[12] "Google Code." URL: http://code.google.com/

[13] "Google Play Store." URL: https://play.google.com/store

[14] "Google Play Store Application Download Statistics." URL: http://en.wikipedia.org/wiki/ Google_Play

[15] Java PathFinder. URL: http://babelfish.arc.nasa.gov/trac/jpf

[16] "JPF and Google Summer of Code." URL: http://babelfish.arc.nasa.gov/trac/jpf/wiki/events/ soc2012

[17] K-9 Mail on Google Play Store. URL: https://play.google.com/store/apps/details?id=com.fsck.k9

[18] P. Ammann and J. Offutt, Introduction to Software Testing. Cambridge University Press, 2008.

[19] "Recycle-locator issue 33." URL: http://code.google.com/p/recycle-locator/issues/detail?id=33

[20] "Robotium test framework". URL: http://code.google.com/p/robotium/

[21] The Java™ Virtual Machine Specification (Java SE 7 Edition). URL: http://docs.oracle.com/javase/specs/ jvms/se7/html/index.html

[22] W. Visser, K. Havelund, G. Brat, and S. Park, "Model checking programs," Proc. Int'l Conf. Automated Soft. Engr., 2000, pp. 3-11.

Appendix A. EHS Model Temporal Rule Collection

Our EHS model currently contains 23 temporal rules, specifying the appropriate calling time of a critical set of event handlers in Android applications. We present and explain them below.

- **Rule 1:** When should an activity component's onCreate() life cycle event handler be called?

$$[True],\ [ACT\_START\_EVENT \wedge \neg ACT\_RUNNING] \Longrightarrow \boldsymbol{X}\ act.onCreate()$$

Explanation: An activity's onCreate() handler should be called next if the activity is not running, and it is requested to be launched.

- **Rule 2 and 3:** When should an activity component's onStart() life cycle event handler be called?

Case 1: $[\boldsymbol{X^{-1}}\ act.onCreate()],\ [\neg ACT\_FINISH\_EVENT] \Longrightarrow \boldsymbol{X}\ act.onStart()$

Case 2: $[\boldsymbol{X^{-1}}\ act.onRestart()],\ [True] \Longrightarrow \boldsymbol{X}\ act.onStart()$

Explanation: An activity's onStart() handler should be called next in two cases. In the first case, it should be called after the activity's onCreate() handler completes as long as the activity is not force to finish. In the second case, it should be called after the activity's onRestart() handler completes.

- **Rule 4 and 5:** When should an activity component's onResume() life cycle event handler be called?

Case 1: $[\boldsymbol{X^{-1}}\ act.onStart()],\ [\neg ACT\_FINISH\_EVENT\ ] \Longrightarrow \boldsymbol{X}\ act.onResume()$

Case 2: $[\boldsymbol{X^{-1}}\ act.onPause()],\ [ACT\_RETURN\_EVENT] \Longrightarrow \boldsymbol{X}\ act.onResume()$

Explanation: An activity's onResume() handler should be called next in two cases. In this first case, the onResume() handler should be called if the previously called event handler was this activity's onStart() handler, and this activity is not forced to finish. In the second case, the onResume() handler should be called if the previously called event handler was this activity's onPause() handler (i.e., users try to pause this activity by switching from it to other activities), and users return to this activity.

- **Rule 6:** When should an activity component's onPause() life cycle event handler be called?

$$[\neg act.onPause()\ \boldsymbol{S_S}\ act.onResume()],\ [ACT\_SWITCH\_EVENT] \Longrightarrow \boldsymbol{X}\ act.onPause()$$

Explanation: An activity's onPause() handler should be called next if it was previously interacting with users, and users now switch to other activities or the home screen.

- **Rule 7:** When should an activity component's onStop() life cycle event handler be called?

$$[\boldsymbol{X^{-1}}\ act.onPause()],\ [ACT\_INVISIBLE\ ] \Longrightarrow \boldsymbol{X}\ act.onStop()$$

Explanation: An activity's onStop() handler should be called next if its onPause() handler was previously called, and this activity becomes invisible (i.e., users did not return to this activity after switching from it).

- **Rule 8:** When should an activity component's onDestroy() life cycle event handler be called?

$$[(\neg act.onRestart()\ \boldsymbol{S_S}\ act.onStop()) \wedge (\neg act.onDestroy()\ \boldsymbol{S_S}\ act.onStop())],\ [ACT\_FINISH\_EVENT\ ] \Longrightarrow \boldsymbol{X}\ act.onDestroy()$$

Explanation: An activity's onDestroy() handler should be called next if the activity was stopped (i.e., no life cycle event handler has been called since its onStop() handler was called), and this activity is now being requested to finish.

- **Rule 9:** When should an activity component's onRestart() life cycle event handler be called?

$$[(\neg act.onRestart()\ \boldsymbol{S_S}\ act.onStop()) \wedge (\neg act.onDestroy()\ \boldsymbol{S_S}\ act.onStop())],\ [ACT\_RETURN\_EVENT] \Longrightarrow \boldsymbol{X}\ act.onRestart()$$

Explanation: An activity's onRestart() handler should be called next if the activity was stopped (i.e., no life cycle event handler has been called since its onStop() handler was called), and users now navigate back to this activity.

- **Rule 10 and 11:** When should a service component's onCreate() life cycle event handler be called?

Case 1: $[True],\ [SER\_START\_EVENT \wedge \neg SER\_RUNNING\ ] \Longrightarrow \boldsymbol{X}\ ser.onCreate()$

Case 2: $[True],\ [SER\_BIND\_EVENT \wedge \neg SER\_RUNNING\ ] \Longrightarrow \boldsymbol{X}\ ser.onCreate()$

Explanation: A service's onCreate() handler should be called next in two cases. In the first case, the onCreate() handler should be called if the service is requested to start, and this service is not running. In the second case the onCreate() handler should be called if the service is requested to start by binding, and this service is not running.

- **Rule 12 and 13:** When should a service component's onStartCommand() life cycle event handler be called?

Case 1: $[\boldsymbol{X^{-1}}\ ser.onCreate()],\ [\neg SER\_FINISH\_EVENT\ \wedge\ SER\_STARTED] \Longrightarrow \boldsymbol{X}\ ser.onStartCommand()$

Case 2: $[True],\ [SER\_START\_EVENT \wedge SER\_RUNNING] \Longrightarrow \boldsymbol{X}\ ser.onStartCommand()$

Explanation: A service's onStartCommand() should be called next in two cases. In the first case, the onStartCommand() handler should be called if (1) the service's onCreate() handler was called previously, (2) the service is launched by normal starting, and (3) the service is not forced to finish. In the second case, the onStartCommand() handler should be called if the service is now requested to start, but it is already running.

- **Rule 14 and 15:** When should a service component's onBind() life cycle event handler be called?

Case 1: $[\boldsymbol{X^{-1}}\ ser.onCreate()],\ [\neg SER\_FINISH\_EVENT\ \wedge\ SER\_BOUND] \Longrightarrow \boldsymbol{X}\ ser.onBind()$

Case 2: $[True],\ [SER\_BINDING\_EVENT \wedge SER\_RUNNING\ ] \Longrightarrow \boldsymbol{X}\ ser.onBind()$

Explanation: A service's onBind() handler should be called next in two cases. In the first case, the onBind() handler should be called if (1) the service's onCreate() handler was previously called, (2) the

service is launched by binding, and (3) the service is not forced to finish. In the second case, the onBind() handler should be called if the service is already running, and another component now requests to bind to it.

- **Rule 16:** When should a service component's onUnbind() life cycle event handler be called?

$$[True],\ [SER\_UNBIND\_EVENT\ \wedge\ SER\_RUNNING] \Longrightarrow \boldsymbol{X}\ ser.onUnbind()$$

Explanation: A service's onUnbind() handler should be called next if the service is running, and another component now requests to unbind to it.

- **Rule 17 and 18:** When should a service component's onDestroy() life cycle event handler be called?

Case 1: $[True],\ [SER\_FINISH\_EVENT\ \wedge\ SER\_STARTED \wedge\ SER\_RUNNING] \Longrightarrow \boldsymbol{X}\ ser.onDestroy()$

Case 2: $[\boldsymbol{X^{-1}}\ ser.onUnbind()],\ [SER\_BOUND \wedge SER\_NO\_BOUND\_CONN] \Longrightarrow \boldsymbol{X}\ ser.onDestroy()$

Explanation: A service's onDestroy() handler should be called next in two cases. In the first case, a running service's onDestroy() handler should be called if the service is launched by normal starting mechanism, and the service is now requested to finish. In the second case, the onDestroy() handler of a service launched by the binding mechanism should be called if the service has no bound clients after the call to its onUnbind() handler.

- **Rule 19:** When should a dynamic message event handler rcv.onReceive() be called?

$$[\neg rcv.unreg()\ \boldsymbol{S_s}\ rcv.reg()],\ [MSG\_EVENT] \Longrightarrow \boldsymbol{X}\ rcv.onReceive()$$

Explanation: A dynamic message event handler rcv.onReceive() should be called next if the receiver *rcv* is properly registered, and *rcv*'s interested message event occurs at this moment.

- **Rule 20:** When should a static message event handler Receiver.onReceive() be called?

$$[True], [MSG\_EVENT] \Longrightarrow \boldsymbol{X}\ Receiver.onReceive()$$

Explanation: A static message event handler should be called next if its interested message event occurs at this moment.

- **Rule 21 to 27:** When should a GUI event handler be called?

$$[(\neg act.onPause()\ \boldsymbol{S_s}\ act.onResume()) \wedge (\neg widget.reg(null)\ \boldsymbol{S_s}\ widget.reg(listener))],\ [GUI\_EVENT] \Longrightarrow \boldsymbol{X}\ listener.onHandleGUIEvent()$$

Explanation: A GUI event handler should be called if (1) the GUI event occurs (including click events, touch events, long click events, menu click events, drag events, hover events, key events), (2) the GUI event listener is properly registered, and (3) the GUI widget's enclosing activity is at foreground.

- **Rule 28:** When should an activity component's onCreateOptionsMenu() handler be called?

  $[(\neg act.onPause()\ \boldsymbol{S_s}\ act.onResume())], [MENU\_CLICK\_EVENT] \Longrightarrow \boldsymbol{X}\ act.onCreateOptionsMenu()$

  Explanation: An activity's onCreateOptionMenu() handler should be called next if the activity is at foreground, and the menu button is clicked.

- **Rule 29:** When should an activity component's onOptionsItemSelected() handler be called?

  $[(\neg act.onPause()\ \boldsymbol{S_s}\ act.onResume())], [MENU\_ITEM\_CLICK\_EVENT]$
  $\Longrightarrow \boldsymbol{X}\ act.onOptionsItemSelected()$

  Explanation: An activity's onOptionsItemSelected() handler should be called next if the activity is at foreground, and a menu item is clicked.

Appendix B. The Collection of Modeled Android APIs

We need to carefully model Android APIs that leverage Android system functionalities or rely on native libraries such that JPF can properly execute an Android application in its JVM and perform verification tasks. Depending on the verification goals, one can choose to model a set of Android APIs while ignore the side effects of other APIs. In this appendix, we list a critical set of Android APIs we have carefully modeled. The modeling took us huge engineering efforts and the list is still expanding.

1. Activity related APIs
   - `public void startActivity(Intent intent)`
   - `public void startActivityForResult(Intent intent, int requestCode)`
   - `public void finish()`
   - `public void finishActivity(int requestCode)`
   - `public final void setResult(int resultCode)`
   - `public void setContentView(int layoutResID)`
   - `public void findViewById(int id)`
   - `public Object getSystemService(String name)`

2. Service related APIs
   - `public ComponentName startService(Intent intent)`
   - `public boolean bindService(Intent intent, ServiceConnection conn, int flags)`
   - `public abstract IBinder onBind(Intent intent)`
   - `public final void stopSelf()`
   - `public final void stopSelf(int startId)`
   - `public final void stopSelfResult(int startId)`

- `public boolean stopService(Intent name)`
- `public void unbindService(ServiceConnection conn)`

3. Broadcast receiver related APIs
   - `public Intent registerReceiver(BroadcastReceiver rcv, IntentFilter filter)`
   - `public Intent registerReceiver(BroadcastReceiver rcv, IntentFilter filter, String permission, Handler scheduler)`
   - `public void unRegisterReceiver(BroadcastReceiver rcv)`
   - `public void sendBroadcast(Intent msg)`
   - `public void sendBroadcast(Intent msg, String permission)`
   - `public void sendOrderedBroadcast(Intent msg, String permission)`
   - `public void sendOrderedBroadcast(Intent intent, String receiverPermission, BroadcastReceiver resultReceiver, Handler scheduler, int initial code, String initialData, Bundle initialExtras)`
   - `public void sendStickyBroadcast(Intent intent)`
   - `public void sendStickyBroadcast(Intent intent, String receiverPermission, BroadcastReceiver resultReceiver, Handler scheduler, int initial code, String initialData, Bundle initialExtras)`

4. GUI related APIs
   - `public void setOnClickListener(View.onClickListener listener)`
   - `public void setOnLongClickListener(View.onLongClickListener listener)`
   - `public void setOnTouchListener(View.onTouchListener listener)`
   - `public void setOnCreateContextMenuListener(View.OnCreateContextMenuListener listener)`
   - `public void setOnDragListener(View.OnDragListener listener)`
   - `public void setOnHoverListenert(View.OnHoverListener listener)`
   - `public void setOnTouchListener(View.OnTouchListener listener)`
   - `public void setOnKeyListener(View.OnKeyListener listener)`

5. Location sensing related APIs
   - `public void requestLocationUpdate(String provider, long minTime, float minDistance, LocationListener listener)`
   - `public void removeUpdates(LocationListener listener)`

- `public boolean enableMyLocation()` (Google maps API in MyLocationOverlay class)
- `public void disableMyLocation()` (Google maps API in MyLocationOverlay class)
- `public List<String> getProviders(boolean enabledOnly)`
- `public List<String> getProviders(Criteria criteria, boolean enabledOnly)`
- `public LocationProvider getProvider(String name)`
- `public List<String> getAllProviders()`
- `public String getBestProvider(Criteria criteria, boolean enabledOnly)`
- `public boolean isProviderEnabled(String providerName)`
- `public Location getLastKnownLocation(String provider)`

6. Wake lock related APIs
   - `public WakeLock newWakeLock(int levalAndFlags, String tag)`
   - `public void acquire()`
   - `public void release()`
   - `public void setKeepScreenOn(boolean keepScreenOn)`

7. Asynchronous task and timer task related APIs
   - `public final AsyncTask<Params, Progress, Result> execute (Params... params)`
   - `public void schedule(TimerTask task, long period)`
   - `public void schedule(TimerTask task, long delay, long period)`
   - `public void schedule(TimerTask task, Date when)`
   - `public void schedule(TimerTask task, long delay)`