Int J Software Informatics, Volume 8, Issue 1 (2014), pp. 21–41
International Journal of Software and Informatics, ISSN 1673-7288

E-mail: ijsi@iscas.ac.cn
http://www.ijsi.org
Tel: +86-10-62661040

CheckerDroid : Automated Quality Assurance for Smartphone Applications

Yepang Liu[1], Chang Xu[2,3], S.C. Cheung[1], and Wenhua Yang[2,3]

[1](Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong, China)

[2](State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China)

[3](Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China)

Abstract Smartphone applications' quality is vital. However, many smartphone applications on market suffer from various bugs. One major reason is that developers lack viable techniques to help expose potential bugs in their applications. This paper presents a practical dynamic analysis tool, CheckerDroid, to help developers automatically detect both functional and non-functional bugs in their Android applications. CheckerDroid currently supports the detection of the following three types of bugs: null pointer exception, resource leak and sensor listener misusage. We built CheckerDroid by extending Java PathFinder (JPF), a widely-used model checker for general Java programs. Our extension addresses two technical challenges. First, Android applications are event-driven and lack explicit control flow information between event handlers. Second, Android applications closely hinge on native framework libraries, whose implementations are platform-dependent. To address these challenges, we derive event handler scheduling policies from Android documentations, and encode them to guide CheckerDroid to realistically execute Android applications. Besides, we modeled the side effects for a critical set of Android APIs such that CheckerDroid can conduct bug detection precisely. To evaluate CheckerDroid, we conducted experiments with seven popular real-world Android applications. CheckerDroid analyzed these applications in a few minutes, and successfully located real bugs in them.

Key words: smartphone applications; quality assurance; functional bugs; energy bugs

Liu Y, Xu C, Cheung SC, Yang W. CheckerDroid: Automated quality assurance for smartphone applications. *Int J Software Informatics*, **Vol.8, No.1 (2014): 21–41. http://www.ijsi.org/1673-7288/8/i181.htm**

1 Introduction

The market of smartphone applications is expanding at an unprecedented rate. As of July 2013, over one million Android applications on Google Play store (i.e., Android official application store) have received 50 billion downloads[13]. Users rely on

This research was partially funded by Research Grants Council (General Research Fund 611813) of Hong Kong, and by National High-Tech Research & Development Program (863 program 2012AA011205), and National Natural Science Foundation (61472174, 91318301, 61321491, 61361120097) of China.
Corresponding authors: Chang Xu, Email: changxu@nju.edu.cn; S.C. Cheung, Email: scc@cse.ust.hk
Received 2013-12-31; Revised 2014-06-19; Accepted 2014-07-28.

these applications for various daily activities such as time scheduling, entertainment, socializing, finance management, and even health care[15]. As such, software quality of these applications becomes vital. Application developers thus are expected to extensively test their applications before releasing them to market.

Unfortunately, the reality is not optimistic. Many applications suffer from different functional and non-functional bugs. A notorious example is that a bug in Android's built-in email application caused thousands of users failing to connect to their mail servers[1]. The pervasiveness of bugs in smartphone applications is attributable to three major reasons. First, many smartphone applications are developed by small teams without dedicated quality assurance. It is not realistic for such small teams to perform a thorough testing of their applications on different devices before releasing these applications. Second, unlike their desktop counterparts, smartphone platforms have a much shorter history. Application developers lack easy-to-use and industrial-strength tools to help analyze their applications for exposing bugs. Existing tools like Robotium[31], although powerful, still require non-trivial manual effort to provide test cases or certain models (e.g., GUI models) for effective analysis. What is even worse, the smartphone applications market is extremely competitive. There are typically hundreds of applications having similar functionalities (e.g., various browsers). Developers need to push their applications or updates (e.g., new features or bug fixes) to market in a short time, as otherwise users can easily switch to other similar products. Thus, effective and efficient tools that can help developers automatically identify potential bugs in their applications are highly desirable.

In this paper, we present a practical bug detection tool, CheckerDroid, to facilitate quality assurance for Android applications. We target Android applications due to their popularity and Android platform's openness. CheckerDroid supports detecting both functional bugs and non-functional bugs. Specifically, CheckerDroid currently is able to detect the following three common patterns of bugs: (1) null pointer exception, (2) resource leak, and (3) sensor listener misusage. We identified these three common bug patterns by a literature survey and an empirical study (Section 3). The first two patterns of bugs can easily lead to application crashes. For example, if an Android application fails to close a database cursor before it is put to background (i.e., a resource leak bug), another application trying to access the same database may crash due to an illegal state exception thrown by the database library[3]. Bugs of the last pattern are non-functional bugs. They can easily lead to energy inefficiency in Android applications. For example, if an Android application does not timely deactivate a sensor when the sensor is no longer being used, the application can continuously acquire sensory data, but these data will not be used for users' benefits (more to be explained in Section 3). With these common bug patterns identified, we in this work propose a set of dynamic analysis algorithms for automated bug detection. These algorithms systematically explore an application's state space for bug detection.

We built CheckerDroid by extending Java PathFinder (JPF), a widely-used dynamic model checker for general Java programs[38]. This extension addressed the following two major technical challenges:

Lack of explicit control flows. First, 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 specify an application's logic in a set of loosely-coupled event handlers. At runtime, these event handlers are implicitly called by the Android system. For example, the onStart() lifecycle handler of an activity component is called after the onCreate() lifecycle handler is called (see Fig. 1 and Section 2 for details), but such calling order is never explicitly specified in the program code. This causes trouble for dynamic analysis tools like JPF as they are designed to execute and analyze programs whose control flows are explicitly stated.

Heavy reliance on native libraries. Android exposes more than 8,000 public APIs to developers[10]. Many of them rely on Android system functionalities or native libraries whose implementations are platform-specific (e.g., thread manipulation and GUI-related APIs). Related code is written in system-native languages (e.g., C and C++), and thus not suitable to be executed in JPF's Java virtual machine. However, the side effect of such code must be considered, as otherwise JPF may encounter various unexpected problems when it executes and analyzes Android applications.

To address the first challenge, we derived event handler scheduling policies from Android specifications, and formulated these policies as an Application Execution Model (AEM model). This model captures application-generic temporal rules that specify the calling relationships between event handlers. Enforcing these rules at runtime can guide JPF to call event handlers at appropriate time. To address the second challenge, we identified a critical set of Android APIs that rely on native libraries, and properly modeled their side effects using JPF's listener and native peer mechanisms (see Section 5 for details). Such API modeling involves non-trivial effort as we will show later. By addressing these two challenges, JPF would be able to execute Android applications such that CheckerDroid could conduct pattern-based analysis for bug detection.

To evaluate CheckerDroid, we conducted controlled experiments by applying CheckerDroid to analyze seven popular real-world Android applications across five different categories. CheckerDroid finished analyzing these applications in only a few minutes, and successfully detected nine real bugs in them.

In summary, we make the following contributions in this paper:

- We propose a dynamic analysis framework to automatically detect three common patterns of functional and non-functional bugs in Android applications.
- We present an application execution model that captures application-generic temporal rules for scheduling event handlers in Android applications. This model is general enough to be used in other Android application analysis techniques.
- We implemented a prototype tool called CheckerDroid and evaluated CheckerDroid using seven popular real-world Android applications. CheckerDroid successfully detected nine real functional bugs and non-functional energy bugs in these applications.

In a preliminary conference version of this work[21], we demonstrated the usefulness of our approach in helping developers locate energy bugs in Android

applications. In this paper, we extend the conference version in four major aspects: (1) studying common patterns of functional bugs in Android applications and designing algorithms for their automated detection (Sections 3 and 4); (2) discussing more details about how we modeled Android library APIs to enable JPF to execute Android applications (Section 5); (3) enhancing our evaluation with more real-world application subjects and result analyses (Section 6); (4) augmenting related work discussions (Section 7).

The rest of this paper is organized as follows. Section 2 introduces the basics of Android applications. Section 3 presents our bug pattern identification study. Section 4 discusses our automated bug detection approach and algorithms. Section 5 gives implementation details of CheckerDroid. Section 6 evaluates our approach and discusses the experimental results. Section 7 reviews representative related work, and finally Section 8 concludes this paper.

2 Background

We select the Android platform for our study because it is currently one of the most widely adopted smartphone platforms and it is open for research. Applications running on Android are primarily written in Java programming language. An Android application is first compiled to Java virtual machine compatible .class files that contain Java bytecode instructions. These .class files are then converted to Dalvik virtual machine executable .dex files that contain Dalvik bytecode instructions. Finally, the .dex files are encapsulated into an Android application package file (i.e., an .apk file) for distribution and installation. For ease of presentation, we in the following may simply refer to "Android application" by "application" when there is no ambiguity. An Android application typically comprises four kinds of components as follows:

Activity. Activities are the only components that contain graphical user interfaces. An application may comprise multiple activities to provide a cohesive user experience.

Service. Services are components that run in the background for conducting long-running tasks like sensor reading. Activities can start and interact with services.

Broadcast receiver. Broadcast receivers define how an application responds to system-wide broadcasted messages. It can be statically registered in an application's configuration file, or dynamically registered at runtime.

Content provider. Content providers manage shared application data, and provide an interface for other components or applications to query or modify these data.

Each application component has a lifecycle defining how it is created, used, and destroyed. Figure 1 shows an activity lifecycle. It starts with a call to onCreate() handler, and ends with a call to onDestroy() handler. An activity's foreground lifetime starts after a call to onResume() handler, and lasts until 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, and the activity would 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.

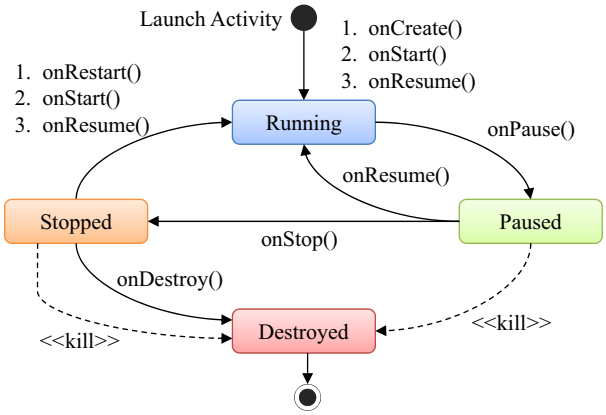

Figure 1. Activity lifecycle diagram

3 Common Bug Pattern Identification

We aim to build a practical tool to help Android application developers automatically detect both functional bugs and non-functional energy bugs in their applications. Learning common bug patterns in real-world Android applications is a key step toward the goal. Therefore, our work started with a study to identify common bug patterns in Android applications. We discuss this study in this section.

Functional bugs have been studied for decades and Android applications are mostly written in the Java programming language. Therefore, we conducted a literature survey to identify common functional bug patterns in general Java applications/programs. Our study revealed two most common patterns of functional bugs. These two bug patterns have been well-studied in literature[19]. They are:

Null pointer exception. Null pointer exceptions happen when a Java program tries to dereference a null object. If an Android application fails to handle such exceptions appropriately, it can easily crash.

Resource leak. Resource leak bugs happen when a Java program fails to properly release certain system resources (e.g., file and database handles) it has acquired during execution before it exits. In Android applications, resource leaks are also common. For example, if an application forgets to close a database connection or cursor before it exits, another application trying to access the same database may fail with exceptions (e.g., IllegalStateException in SQLite database[3]).

However, for non-functional energy bugs, we have very limited understanding since these bugs are relatively new. So we conducted an empirical study to understand real-world energy bugs in Android applications and identify common energy bug patterns. For our empirical study, we randomly collected 174 open-source Android applications from three primary open-source software hosting platforms, i.e., Google Code[12], GitHub[11], and SourceForge[35]. By studying these 174 applications, we found that 33 of them have confirmed or fixed energy bugs. We then carefully investigated these energy bugs by checking all related data in the concerned application's source repository. These data include energy bug reports, developers' and users' comments on bug reports, bug-fixing patches, patch reviews,

and commit logs of bug-fixing revisions. From this empirical study, we obtained two major findings. First, developers found it difficult to diagnose energy bugs. To expose or reproduce these bugs, they need to conduct extensive testing of the affected applications on certain devices and perform energy profiling at the same time. To figure out the root causes of these bugs, they need to instrument the affected applications and collect runtime information for analysis. Such a process is time-consuming and tedious. Our second finding is related to energy bug patterns. We found that *although the root causes of energy bugs can be application specific, many of them are closely related to misuse of sensors.* Specifically, we observed the following common pattern of energy bugs:

Sensor listener misusage. To use a sensor, an Android application needs to register a sensor listener with the Android system, and specify a sensing rate[2]. A sensor listener defines how an application reacts to sensor value or status changes. When a sensor is no longer needed, its corresponding listener should be unregistered. Forgetting to unregister would lead to wasted sensing operations and battery energy.

4 Automated Bug Detection Approach

With common bug patterns identified, we in this section propose our automated bug detection approach. We start with an overview of our approach.

4.1 Approach overview

Our approach is based on dynamic program analysis. Figure 2 shows its high-level abstraction. It takes the Java bytecodes and configuration files of an Android application under analysis as inputs. The Java bytecodes define the application's program logic, and can be obtained by compiling its source code or transforming its Dalvik bytecodes[24]. The configuration files specify the application's components, GUI layouts and so on. The general idea of our approach is that our runtime controller executes an Android application in JPF's Java virtual machine (JVM)[1] with different inputs, and systematically explores its application states. During each execution, our bug detectors monitor the application's running states (e.g., heap information and sensor registration/unregistration; see Section 5 for more details) and conducts pattern-based analysis to detect corresponding patterns of bugs discussed earlier. When all executions finish, our approach will summarize all detected bugs and report actionable information for helping debugging.

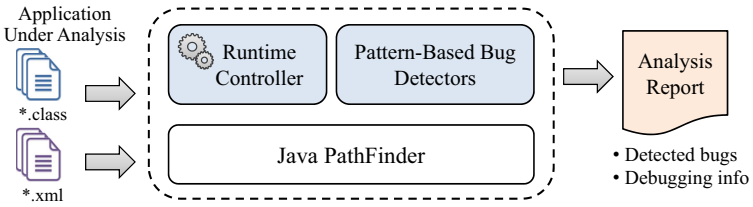

Figure 2. Approach overview

This high-level abstraction looks intuitive, but some challenging questions

[1]On real devices, an Android application runs in a registered-based Dalvik VM, while JPF's JVM is stack-based. This difference does not affect our analysis.

remain unanswered: How can JPF realistically execute an Android application in its JVM? How to generate inputs such that JPF can systematically explore an Android application's state space? How to automatically detect the three common patterns of bugs? We answer these questions below.

4.2 Dynamic execution of android applications

An Android application starts with its main activity, and ends after all its components are destroyed. It keeps handling received events (including user interaction events and system 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 or global program data. We thus use the sequence of event handlers that have been called to represent an *application state*. From this discussion, we can see that in order to simulate real executions of Android applications, we need to address the following two issues: (1) generating user interaction events to drive an application's execution, and (2) deriving event handler scheduling policies from Android specifications, and leveraging them for guiding the runtime scheduling of event handlers. We now discuss how we address these two issues below.

User interaction event sequence generation. In Android applications, developers need to provide a GUI layout configuration file for each activity component in their applications. This practice facilitates the generation of user interaction events. Conceptually, our generation process contains two parts: static and dynamic. In the static part, i.e., before executing an application, our runtime controller first analyzes the application's GUI layout configuration files to learn the GUI model of each activity component (recall that only activities have GUIs). These GUI models map each GUI element (e.g., a button) of an activity component to a set of possible user actions (e.g., button clicks). Then, in the dynamic part, i.e., when executing an application, our runtime controller monitors the application's application state. When the application waits for user interactions (e.g., after an activity's onResume() handler is called), our controller would generate required events and feed them to the foreground activity for handling by analyzing the activity's GUI model. This generation is done in an exhaustive way by enumerating all possible events associated with each activity component. Our controller keeps doing so until (1) the length of a generated event sequence reaches a pre-specified upper bound,[2] or (2) the analysis time reaches a pre-specified limit (i.e., time budget), or (3) the application exits. We need to set an upper bound to the length of generated event sequences or a time budget for the analysis because an application's state space can be unbounded (i.e., users can interact with an application in infinite ways). Without such an upper bound or time budget, an analysis tool may not be able to finish analysis in a timely fashion or it can easily exhaust computational resources (e.g., RAM). In this way, our runtime controller generates all possible event sequences bounded by a length limit to explore different application states.

[2]In practice, this upper bound can be configured in an interactive way. For example, the tool user can set a relatively small upper bound initially and check whether the resulting code coverage and detected issues are satisfactory or not. If not, a larger upper bound can be configured.

Event handler scheduling. For guiding the runtime scheduling of event handlers, we derive an application execution model (or AEM) from Android specifications. Our AEM model is a collection of temporal rules. They are application-generic and should be enforced at runtime (unary temporal connective G means "always"):

$$AEM := G \bigwedge_i R_i$$

Each temporal rule is expressed in the following form:

$$R_i := [\psi], [\phi] \Rightarrow \lambda$$

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 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. The whole rule means: If both ψ and ϕ hold, λ should be executed next.

We list some example temporal rules below. Propositional connectives $\wedge$, $\Rightarrow$, and $\neg$ in these examples follow their traditional interpretations, and temporal connectives are explained as follows. Unary temporal connective X means "next", and its past time analogue X^{-1} means "previously". Binary temporal connective S means "since". Specifically, a temporal formula "$F_1\ S\ F_2$" means that F_2 held at some time in the past, and since then F_1 always holds.

– Example 1: When to call the lifecycle handler onStart() of an activity *act*?

$$[\boldsymbol{x}^{-1}\, act.onCreate()], [\neg ACT\_FINISH\_EVENT] \Rightarrow \boldsymbol{X}\, act.onStrat()$$

– Example 2: When to call a button-click GUI event handler onClick()?

$$[(\neg act.onPause() \boldsymbol{S} act.onResume()) \wedge (\neg btn.reg(null) \boldsymbol{S} btn.reg(listener))],$$
$$[BTN\_CLICK] \Rightarrow \boldsymbol{X}\, listener.onClick()$$

The first rule requires an activity's onStart() handler to be called after its onCreate() handler completes as long as the activity is not forced to finish. The second rule requires a button-click event handler to be called if: (1) the button is clicked, (2) its enclosing activity is at 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 registered. For the entire collection of the temporal rules, readers may refer to the appendix of this paper.

Our AEM model is converted to a decision procedure to decide which event handlers to be called according to an application's execution history and its newly received events. By this, we are now able to execute an application a finite number of times in JPF's virtual machine and explore different application states for bug detection.

4.3 Bug detection algorithms

Since functional bugs have been studied for a long time, we can adapt existing standard algorithms for the detection of these bugs[19]. Specifically, our algorithms for detecting null pointer exception and resource leak bugs work as follows:

Detecting null pointer exception bugs. To detect null-pointer exception bugs, our bug detector actively monitors each object dereference operation during the execution of an Android application. Specifically, it intercepts the execution of a subset of Java bytecode instructions that involves object reference resolving (e.g., the bytecode *invokevirtual* invokes a virtual method on an object *obj*). After bytecode interception, our bug detector checks if the concerned object reference equals null or not. If yes, it reports a warning.

Detecting resource leak bugs. To detect resource leak bugs, our bug detector tracks the resource acquisition and releasing operations during the execution of an Android application. It then checks for the violation of the following resource safety policy[5,36]: once an application acquires certain system resource, it needs to properly release the resource before it exits.

Detecting sensor listener misusage bugs. To detect sensor listener misusage bugs, we reduce the bug pattern to resource leak. The idea is that we can treat the sensor listeners as finite system resource. Then, sensor listener registration/unregistration operations can be viewed as resource acquisition/release operations. By this reduction, we can apply the above resource leak detection algorithm to detect sensor listener misusage bugs.

5 CheckerDroid Implementation

We implemented our approach as a prototype tool called CheckerDroid on top of JPF. We clarify important implementation details of CheckerDroid in this section.

First, besides tracking the standard JPF program state information (i.e., call stack of each thread, heap and scheduling information), CheckerDroid also maintains the following runtime information: (1) a stack of active activities, their lifecycle status, and visibility of their containing GUI elements, (2) a list of running services and their lifecycle status, and (3) a list of registered broadcast receivers.

Second, as mentioned earlier, many Android APIs rely on Android system functionalities or native libraries. This causes a big trouble to JPF because the bytecode instructions of these Android APIs are typically not available for analysis[23]. Even if these bytecode instructions can be obtained (e.g., by building the Android framework), JPF will still fail to execute them because it cannot handle the transitively called native methods. Due to these reasons, we need to properly model Android APIs and their side effects. We note that completely and precisely modeling all APIs requires enormous engineering effort. As such, in our work, we took a pragmatic approach by modeling a critical set of APIs, which are commonly called in Android applications. We give concrete examples below for illustrating how we model some typical APIs.

Activity and service API modeling. Android applications can start new activities by calling several APIs (e.g., startActivity()). After completing certain tasks, an active activity can also be finished by calling corresponding APIs. Here, we introduce the modeling of startActivity() API as an example. The major side effect

of this API is to switch the current foreground activity to background, launch the new activity, and put it to foreground. In order to model such side effects, we guide JPF to conduct the following tasks[3], as illustrated in Fig. 3:

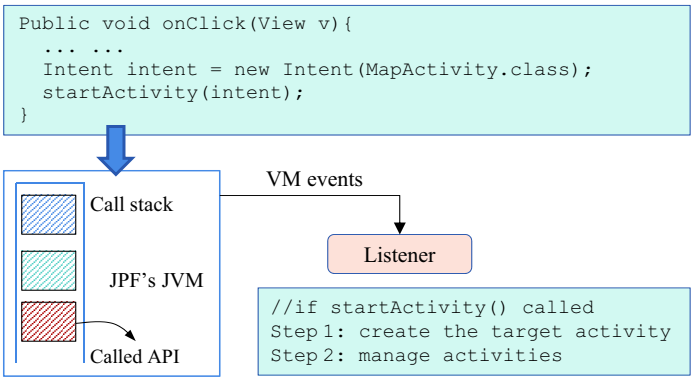

Figure 3. Modeling the startActivity() API

- **API call interception.** We closely monitor an application's state changes, and intercept each call to the startActivity() API.
- **Side effects abstraction.** The startActivity() API leverages system level functionality (e.g., thread manipulation), and relies on native libraries (e.g., native window manager). We ignore the real implementations of the API and concerned native libraries, and abstract its side effects.
- **Activity management.** After call interception and native peer creation, we can manage the activity stack maintained by our CheckerDroid to switch the current foreground activity to background and put the new activity to foreground.

The first two tasks can be done by registering a listener to monitor the JPF's JVM events and creating a stub native peer for the startActivity() API. The third task can be done by directly manipulating the call stack of JPF's JVM to invoke certain lifecycle event handlers of the concerned activities. Manipulating the call stack of JPF's JVM is needed because an application under analysis runs in JPF's JVM, instead of the host JVM, where JPF and its listeners run.

Similar to activities, other application components can start or stop a service component by calling certain APIs. The modeling of these APIs resembles the modeling of activity APIs. We do not make further elaborations in this paper.

Broadcast receiver API modeling. In Android applications, broadcast receivers can be statically registered in application configuration files or dynamically registered by other components (typically, activities or services) at runtime by calling broadcast receiver APIs. A dynamically registered broadcast receiver can be unregistered by calling unregistration APIs. Each broadcast receiver is associated with a message filter, which specifies its interested message types. At runtime, other application components can broadcast messages by calling the broadcasting APIs.

[3]The modeling of other APIs generally follow this three-step approach.

As discussed earlier, CheckerDroid maintains a list of registered broadcast receivers (static receivers are considered as always registered). Figure 4 illustrates how such management is done, as well as how the concerned APIs are modeled. 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 API calls. When a dynamic broadcast receiver is registered, our receiver manager will add this receiver to the dynamic receiver list. Similarly, when a dynamic broadcast receiver is unregistered, it will be removed from the list. By such management, CheckerDroid can then properly pick up the appropriate receivers to handle broadcasted messages at runtime by checking the message types. Similar to the activity API modeling, all management methods (e.g., receiver adding method) are called by directly manipulating JPF's JVM.

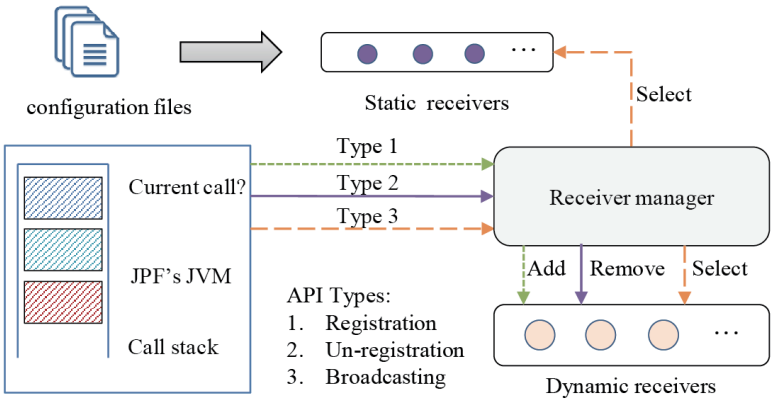

Figure 4. Broadcast receiver API modeling

GUI-related API modeling. GUIs play a central role in Android applications. However, their construction and manipulation highly rely on native graphics libraries. This complicates the analysis of Android applications using JPF as JPF cannot analyze native code. Therefore, we need to properly model GUI-related APIs. In the following, we discuss the modeling of a commonly called API findViewById() as an example. The modeling of other GUI-related APIs is similar. The findViewById() API traverses the GUI element tree of a certain activity's GUI, and locates a GUI element with a specific ID. Then the application can perform legitimate operations on the retrieved GUI element. For example, the following code snippet locates a button and registers a click event listener with it.

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

Figure 5 illustrates how we model the findViewById() API. Similar to our earlier user interaction event generation idea, the modeling consists of two parts: a static part and a dynamic part. In the static part, CheckerDroid pre-analyzes an application's configuration files to learn the GUI model of each activity component. The GUI

model contains key information such as each GUI element's type, ID, its associated text, and legitimate user events on it. Then, at runtime, CheckerDroid monitors the call to findViewById(). When it is called, CheckerDroid would identify the current activity and analyze its GUI model. By doing so, CheckerDroid would be able to find all necessary information about the GUI element under search. Finally, CheckerDroid can create a corresponding object in JPF's JVM for the GUI element if it has not been constructed.

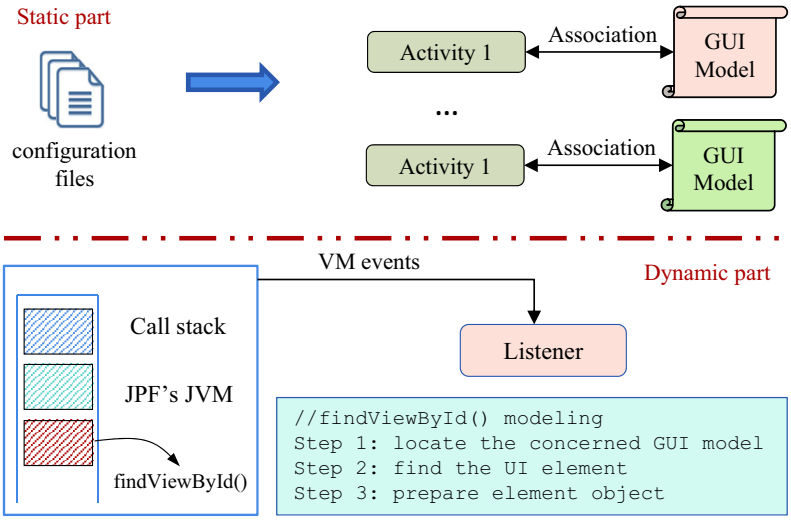

Figure 5. Modeling findViewById() API

From the above discussions, we can see that modeling Android APIs is labor-intensive. It took us several months to model a critical subset of Android APIs[14]. In practice, to save manual effort, one can choose to ignore the side effects of some APIs if these effects are not relevant to the verification target. This "partial modeling" helps reduce the state space that JPF needs to explore.

6 Evaluation

In this section, we evaluate our approach by controlled experiments. We aim to answer the following two research questions:

- **RQ1 (Effectiveness):** *Can our approach effectively detect both functional bugs and non-functional energy bugs in real-world Android applications?*
- **RQ2 (Efficiency):** *What is the analysis overhead of our approach? Is it practical to apply our approach to analyze real-world large-scale Android applications?*

6.1 Experimental setup

We selected seven popular open-source Android applications as our experimental subjects. Table 1 lists their basic information, including: (1) application name, (2) application size, (3) number of downloads on market, (4) application category, and

(5) source repository hosting websites. We can observe from the table that these applications come from 6 different categories and have been popularly downloaded on market. Besides, the application size ranges from small (e.g., Recycle-locator), medium (e.g., Ushahidi) to large (e.g., C:geo). All applications were compiled for Android 2.3.3. We choose Android 2.3.3 because it is one of the most widely-adopted Android platforms and compatible with most applications on market. We conducted our experiments on a dual-core machine with Intel Core i5 CPU and 8GB RAM, running Windows 7. In the following subsections, we study our research questions accordingly.

Table 1 Application subjects information

Application name	Size (LOC)	Downloads	Application category	Source availability
Omnidroid	12.4 K	1 K ∼ 5 K	Productivity	Google Code
DroidAR	18.1 K	1 K ∼ 5 K	Tools	Google Code
Recycle-locator	3.2 K	1 K ∼ 5 K	Travel & Local	Google Code
C:geo	27.0 K	1 M ∼ 5 M	Entertainment	GitHub
OI File Manager	6.7 K	5 M ∼ 10 M	Productivity	GitHub
AnySoftKeyboard	19.3 K	500 K ∼ 1 M	Tools	GitHub
Ushahidi	10.2 K	5 K ∼ 10 K	Communication	GitHub

6.2 Effectiveness of CheckerDroid

To answer research question RQ1, we ran CheckerDroid to analyze each of our application subject for bug detection. We controlled CheckerDroid to generate at most six user interaction events during each application execution. This suffices for CheckerDroid to explore a large number of application states. We examined CheckerDroid's top ranked analysis reports to see whether they helped locate real bugs in these applications. We report the results below.

Encouragingly, CheckerDroid successfully detected nine real bugs in our application subjects. Table 2 lists the information of these detected bugs, including their bug report IDs and bug patterns. For functional bugs, CheckerDroid detected four null pointer exceptions and three resource leaks. For non-functional energy bugs, CheckerDroid detected two sensor listener misusage bugs. We explain some of these detected bugs below for illustration.

Table 2 Bugs detected by CheckerDroid and bug detection overhead

Application name	Detected bugs		Analysis overhead	
	Bug report ID	Bug pattern	Time (Second)	Memory (MB)
DroidAR	27	Sensor listener misusage	311	272
Recycle-locator	33	Sensor listener misusage	52	178
C:geo	124	Null pointer exception	183	395
OI File Manager	30	Null pointer exception	34	127
AnySoftKeyboard	80	Null pointer exception	106	259
Ushahidi	100	Resource leak	35	188
	46	Resource leak		
Omnidroid	77	Null pointer exception	284	447
	103	Resource leak		

For example, DroidAR is a framework for augmented reality on Android. It leverages sensory data to digitalize the real world and make users' environment interactive. CheckerDroid detected that its location listener is never unregistered after usage. This poses big threats to a battery's lifetime as these sensors may keep running until the application process is killed[4]. Another example is the null pointer exception in the application OI File Manager, which can help Android users manage files of various formats. The exception happens when users select some documents in the root directory and click "compress" button to produce an archive file. Then, OI File Manager would crash because of this unhandled runtime exception. Therefore, from the above discussion, we derive our answer to research question RQ1 is: *our approach can effectively help developers detect both functional bugs and non-functional energy bugs in their applications.*

6.3 Efficiency of CheckerDroid

To answer research question RQ2, we recorded the analysis overhead of CheckerDroid when applying it to analyze each of our application subjects for bug detection. Table 2 (the right part) presents the overhead results, including analysis time and memory consumption. These results were averaged over three different runs.

We can make two observations from Table 2. First, all analyses finished in a few minutes and the memory consumption is less than 600 MB. Second, for our three largest application subjects C:geo, AnySoftKeyboard, and DroidAR, their analysis overhead is also affordable. For example, the analysis of C:geo, which contains 27 thousand lines of code, finished in around three minutes and the consumed memory is less than 400 MB. Therefore, from these results, we derive our answer to research question RQ2: *the analysis overhead of* CheckerDroid *can be well-supported by modern PCs, and it is practical to apply* CheckerDroid *to analyze real-world large Android applications.*

6.4 Discussion

Our approach is independent of its underlying program analysis framework. Currently, we implemented it on top of JPF because there is no other suitable tool but JPF is a highly extensive Java program verification framework (Android programs are written in Java). Still, analyzing Android applications using JPF is challenging. First, we need to derive an extensible AEM model from Android specifications and enforce it to make JPF call event handlers in a reasonable way. Second, some Android APIs rely on native libraries whose implementation is specific to hardware and operating systems. The semantics of these APIs and their enclosing library classes have to be properly modeled due to JPF's closed-world assumption. Modeling Android libraries is known to be a difficult and tedious task[23]. Our current implementation only considered a critical subset of library classes and concerned APIs and can already analyze many real-world Android applications[14]. Extending our tool to support more Android APIs is possible and we are exactly on this way.

Our work has some limitations. First, our approach currently cannot simulate complex user inputs such as gestures. We will study the effects of this limitation in

future. Second, CheckerDroid can generate false alarms, especially when detecting null-pointer dereference defects.[4] We analyzed corresponding warnings and realized the false alarms mostly arise from the incomplete and imprecise modeling of Android APIs. Although related studies[23] modeled certain APIs using simple stubs (e.g., randomly returning a value at call boundaries), our experience from the evaluation with real-world subjects suggests that the quality of API models can affect certain analysis. So we believe these false alarms can be removed with more complete and precise modeling of Android APIs,[5] which only requires engineering efforts. Besides, we are not clear whether our approach can be easily generalized to help diagnose other types, especially unknown types of functional and non-functional bugs. We also make this our future work.

7 Related Work

Our work relates to several research topics, including null pointer exception detection, resource leak detection, and energy efficiency analysis. We discuss some representative work in this section.

Null pointer exception detection. Null pointer exception commonly occurs in Java programs, and it significantly hurts software reliability. Many static analysis tools, including FindBugs[9], ESC/Java[8], PMD[30], JLint[17], Soot[33], and Lint4J[20], can identify null pointer exception. To detect null pointer bugs, FindBugs contains a specifically designed analyzer, which is neither sound nor complete. ESC/Java tries to find all violations to a specified null/non-null annotation, which is manually provided by developers. For a statement that dereferences a variable, Soot can determine whether the variable could be null or not. These tools mostly use static analysis techniques to detect null pointer exception based on generic defect patterns, and usually report too many false positives or negatives. Sinha et al. present an approach for locating and repairing faults that cause null pointer exceptions in Java programs[32]. They use the dynamic stack trace to avoid the imprecision problems of static analysis. Spoto et al.[34]. and Hovemeyer et al.[16]. both use annotations to improve the precision of their null pointer analysis.

Resource leak detection. System resources are finite, and developers have to ensure acquired resources to be released eventually. This task is error-prone. Empirical evidence shows that resource leaks commonly occur[39]. Researchers proposed language-level mechanisms and automated management techniques to prevent such leaks[7]. Various tools were also designed to detect resource leaks[5,36]. For example, QVM[5] is a specialized runtime environment for detecting defects in Java programs. It monitors application executions and checks for violations of resource safety policies. Tracker[36] is an industrial-strength tool for finding resource leaks in Java programs. It conducts inter-procedural static analysis to ensure no resource safety policy is violated on any execution path. The major

[4] We carefully sampled the warnings reported by our tool and found that our resource leak and sensor listener misusage bug detectors are quite precise, but the null pointer bug detector could generate around 38% false alarms (we randomly sampled 50 null pointer exception warnings and confirmed 19 of them as false alarms).

[5] Another viable solution to remove false alarms is to implement our approach on real Android platform by modifying its software stack (e.g., Dalvik virtual machine). In this case, all application executions will be real executions and all detected bugs will be true bugs.

difference between our work and these pieces of work is that our detection technique works for event-driven Android applications. We actively scheduled event handlers to drive application executions, and addressed the technical challenges in user event generation and Android native library modeling, which are required when analyzing Android applications in popular program analysis frameworks like JPF.

Energy efficiency analysis. In the past several years, researchers proposed various techniques to help improve smartphone applications' energy efficiency. Kim et al. proposed to use power signatures based on system hardware states to detect energy-greedy malwares[37]. Pathak et al. characterized energy bugs in smartphone applications[29]. They also proposed eProf to help estimate an application's energy consumption by tracking the activities of energy-consuming entities when an application runs on mobile devices[28]. WattsOn[22] shares a similar spirit with eProf but enables energy emulation on the developers' workstations. MAUI[6] helped offload "energy-consuming" tasks to resource-rich infrastructures. EnTracked[18] and RAPS[25] adopted different heuristics to guide an application to use GPS sensors in a smart way. Little Rock[26] suggested a low-power processor for energy-consuming sensing operations. SALSA[27] helped select optimal data links for saving energy in large data transmission. The energy bug detection part of our work shares a similar goal with these discussed pieces of work, but focuses on detecting energy bugs by systematically exploring different application states. Our work incurs reasonable overhead, and provides developers with actionable information to diagnose detected energy bugs.

8 Concluding Remarks

In this paper, we have presented a practical tool, CheckerDroid, to help Android application developers automatically detect both functional and non-functional bugs in their applications. CheckerDroid supports the detection of three common patterns of bugs: (1) null pointer exception, (2) resource leak, and (3) sensor listener misusage. We built CheckerDroid by extending Java PathFinder, a popular model checker for Java[38]. The extension addressed several technical challenges including event handler scheduling and Android library modeling. To evaluate CheckerDroid, we conducted controlled experiments using popular real-world Android applications. Our evaluation results confirmed that CheckerDroid can effectively and efficiently help developers detect real bugs and identify quality improvement opportunities in their Android applications.

In our ongoing work, we are investigating more real-world Android applications to learn more common bug patterns to further extend the capability of CheckerDroid. We believe that our work together with other related ones will help improve the quality of smartphone applications, and this can benefit millions of users.

Acknowledgement

This research was partially funded by Research Grants Council (General Research Fund 611813) of Hong Kong, and by National High-Tech Research & Development Program (863 program 2012AA011205), and National Natural Science Foundation (61472174, 91318301, 61321491, 61361120097) of China.

References

[1] Android Issue 61785. https://code.google.com/p/android/issues/detail?id=61785

[2] Android Sensor Management. http://developer.android.com/reference/android/hardware/SensorManager.html

[3] Android SQLite Database. http://developer.android.com/reference/android/database/sqlite/

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

[5] Arnold M, Vechev M, Yahav E. QVM: an efficient runtime for detecting defects in deployed systems, ACM Trans. Software Engineering and Methodology, 2011, 21(2): 1–2, 35.

[6] Cuervo E, Balasubramanian A, Cho D, Wolman A, Saroiu S, Chandra R, Bahl P. MAUI: making smartphones last longer with code offload. Proc. Int'l Conf. Mobile Systems, Applications, and Services (MobiSys 10). ACM. 2010. 49–62.

[7] Dillig I, Dillig T, Yahav E, Chandra S. The CLOSER: automating resource management in Java. Proc. Int'l Symp. Memory Management (ISMM 08). ACM. 2008. 1–10.

[8] ESC/Java. http://en.wikipedia.org/wiki/ESC/Java

[9] FindBugs. http://findbugs.sourceforge.net/

[10] Felt AP, Chin E, Hanna S, Song D, Wagner D. Android permission demystified. Proc. ACM Conf. Computer and Communications Security. 2011. 627–638.

[11] GitHub website. https://github.com/

[12] Google Code website. https://code.google.com/

[13] Google Play Wiki Page. http://en.wikipedia.org/wiki/Google_Play

[14] GreenDroid website. http://sccpu2.cse.ust.hk/greendroid/

[15] How People Really Use Mobile. Harvard Business Review, Jan. 2013.

[16] Hovemeyer D, Spacco J, Pugh W. Evaluating and tuning a static analysis to find null pointer bugs. Proc. 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 05). 2005. 13-19.

[17] JLint. http://jlint.sourceforge.net/

[18] Kjærgaard MB, Langdal J, Godsk T, Toftkjær T. EnTracked: energy-efficient robust position tracking for mobile devices. Proc. Int'l Conf. Mobile Systems, Applications, and Services (MobiSys 09). ACM. 2009. 221–234.

[19] Liang G, Wang Q, Xie T, Mei H. Inferring project-specific bug patterns for detecting sibling bugs. Proc. ACM SIGSOFT Symp. Foundations of Softw. Engr. (FSE 13). 2013. 565–575.

[20] Lint4j. http://www.jutils.com/

[21] Liu Y, Xu C, Cheung SC. Where has my battery gone? Finding sensor related energy black holes in smartphone applications. Proc. IEEE Int'l Conf. on Pervasive Computing and Communications (PerCom 13). 2013. 2–10.

[22] Mittal R, Kansal A, Chandra R. Empowering developers to estimate app energy consumption. Proc. 18th Int'l Conf. Mobile Computing and Networking (Mobicom 12). 2012. 317–328.

[23] Mirzaei N, Malek S, Păsăreanu CS, Esfahani N, Mahmood R. Testing android apps through symbolic execution. SIGSOFT Softw. Eng. Notes, 2012, 37: 1–5.

[24] Octeau D, Jha S, McDaniel P. Retargeting Android applications to Java bytecode. Proc. ACM SIGSOFT Int'1 Symp. Foundations of Soft. Engr. (FSE 12). ACM. 2012.

[25] Paek J, Kim J, Govindan R. Energy-efficient rate-adaptive GPS-based positioning for smartphones. Proc. Int'l Conf. Mobile Systems, App., and Services (MobiSys 10). ACM. 2010. 299–314.

[26] Priyantha B, Lymberopoulos D, Liu J. LittleRock: Enabling Energy-Efficient Continuous Sensing on Mobile Phones. IEEE Pervasive Computing, 2011, 10: 12–15.

[27] Ra M, Paek J, Sharma AB, Govindan R, Krieger MH, Neely MJ. Energy-delay tradeoffs in smartphone applications. Proc. Int'l Conf. Mobile Systems, Applications, and Services (MobiSys 10). ACM. 2010. 255–270.

[28] Pathak A, Hu YC, Zhang M. Where is the energy spent inside my app? Fine grained energy accounting on smartphones with Eprof. Proc. Euro. Conf. Comp. Sys. (EuroSys 12). 2012. 29–42.

[29] Pathak A, Jindal A, Hu YC, Midkiff SP. What is keeping my phone awake? Characterizing and detecting no-sleep energy bugs in smartphone apps. Proc. 10th Int'l Conf. Mobile Systems, Applications, and Services (MobiSys 12). 2012. 267–280.
[30] PMD. http://pmd.sourceforge.net/
[31] Robotium, a testing framework for Android applications. http://code.google.com/p/robotium/.
[32] Sinha S, Shah H, Gorg C, Jing S, Kim M, Harrold MJ. Fault localization and repair for Java runtime exceptions. Proc. Int'l Symp. Software Testing and Analysis(ISSTA 09). 2009. 153–164.
[33] Soot. http://www.sable.mcgill.ca/soot/
[34] Spoto F. Precise null-pointer analysis. Softw. Syst. Model., May 2011, 10(2): 219–252.
[35] SourceForge website. http://sourceforge.net/
[36] Torlak E, Chandra S. Effective interprocedural resource leak detection Proc. Int'l Conf. Soft. Engr. (ICSE 10). 2010. 535–544.
[37] Kim H, Smith J, Shin KG. Detecting energy-greedy anomalies and mobile malware variants. Proc. Int'l Conf. Mobile Sys., App's, and Services (MobiSys 08). 2008. 239–252.
[38] Visser W, Havelund K, Brat G, Park S. Model checking programs. Proc. Int'l Conf. Automated Soft. Engr(ASE 00). 2000. 3–11.
[39] Weimer W, Necula GC. Finding and preventing run-time error handling mistakes. Proc. ACM SIGPLAN Conf. Object-oriented Prog., Sys, Lang., and App's (OOPSLA 04). 2004. 419–431.

Appendix 1: Temporal rules in our AEM model

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

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

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

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

Case 1:

$$[\boldsymbol{X}^{-1} act.onCreate()], [\neg ACT\_FINISHING\_EVENT] \Rightarrow \boldsymbol{X}\, act.onStart()$$

Case 2:

$$[\boldsymbol{X}^{-1} act.onRestart], [True] \Rightarrow \boldsymbol{X}\, act.onStart()$$

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 forced to finish. In the second case, it should be called after the activity's onRestart() handler completes.

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

Case 1:

$$[\boldsymbol{X}^{-1} act.onStart()], [\neg ACT\_FINISHING\_EVENT] \Rightarrow \boldsymbol{X}\, act.onResume()$$

Case 2:

$$[\boldsymbol{X}^{-1} act.onPause()], [\neg ACT\_RETURN\_EVENT] \Rightarrow \boldsymbol{X}\, act.onResume()$$

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?

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

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?

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

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?

$$[(\neg act.onRestart() \boldsymbol{S} act.onStop()) \wedge (\neg act.onDestroy() \boldsymbol{S} act.onStop())],$$
$$[ACT\_FINISHING\_EVENT] \Rightarrow \boldsymbol{X} act.onDestroy()$$

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?

$$[(\neg act.onRestart() \boldsymbol{S} act.onStop()) \wedge (\neg act.onDestroy() \boldsymbol{S} act.onStop())],$$
$$[ACT\_FINISHING\_EVENT] \Rightarrow \boldsymbol{X} act.onRestart()$$

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.

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

Case 1:

$$[True], [SERVICE\_START\_EVENT \wedge \neg SERVICE\_RUNNING\_EVENT]$$
$$\Rightarrow \boldsymbol{X} ser.onCreate()$$

Case 2:

$$[True], [SERVICE\_BIND\_EVENT \wedge \neg SERVICE\_RUNNING\_EVENT]$$
$$\Rightarrow \boldsymbol{X} ser.onCreate()$$

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.

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

Case 1:

$$[\boldsymbol{X}^{-1} ser.onCreate()], [\neg SERVICE\_FINISH\_EVENT \wedge SERVICE\_STARTED] \Rightarrow \boldsymbol{X}\, ser.onStartCommand()$$

Case 2:

$$[True], [SERVICE\_START\_EVENT \wedge SERVICE\_RUNNING] \Rightarrow \boldsymbol{X}\, ser.onStartCommand()$$

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.

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

Case 1:

$$[\boldsymbol{X}^{-1} ser.onCreate()], [\neg SERVICE\_FINISH\_EVENT \wedge SERVICE\_BOUND] \Rightarrow \boldsymbol{X}\, ser.onBind()$$

Case 2:

$$[True], [SERVICE\_BINDING\_EVENT \wedge SERVICE\_RUNNING] \Rightarrow \boldsymbol{X}\, ser.onBind()$$

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?

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

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

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

Case 1:

$$[True], [SERVICE\_FINISH\_EVENT \\ \wedge SERVICE\_WAS\_STARTED \wedge SERVICE\_RUNNING] \Rightarrow \boldsymbol{X}\, ser.onDestroy()$$

Case 2:

$$[\boldsymbol{X}^{-1} ser.onUnbind()], [SERVICE\_WAS\_BOUND \\ \wedge SERVICE\_HAS\_NO\_BOUND\_CONNECTIONS] \Rightarrow \boldsymbol{X}\, ser.onDestroy()$$

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?

$$[\neg rcv.unreg() \boldsymbol{S} rcv.unreg()], [MSG\_EVENT] \Rightarrow \boldsymbol{X}\, rcv.onReceive()$$

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?

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

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

Rules 21-27: When should a GUI event handler be called?

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

Explanation: A GUI event handler should be called if (1) the GUI event occurs (e.g., click events, touch events etc.), (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?

$$[\neg act.onPause() \boldsymbol{S} act.onResume()], [MENU\_CLICK\_EVENT] \\ \Rightarrow \boldsymbol{X}\, act.onCreateOptionMenu()$$

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?

$$[\neg act.onPause() \boldsymbol{S} act.onResume()], [MENU\_ITEM\_CLICK\_EVENT] \\ \Rightarrow \boldsymbol{X}\, act.onOptionItemSelected()$$

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