A lightweight approach to component-level exception mechanism for robust Android apps

Kwanghoon Choi\textsuperscript{a,1}, Byeong-Mo Chang\textsuperscript{b,*,2}

\textsuperscript{a} Computer & Telecommunication Engineering Division, Yonsei University, Wonju, Gangwon-Do 26493, Republic of Korea
\textsuperscript{b} Department of Computer Science, SooMyung Women's University, Seoul 04310, Republic of Korea

\textbf{Abstract}

Recent researches have reported that Android programs are vulnerable to unexpected exceptions. One reason is that the current design of Android platform solely depends on Java exception mechanism, which is unaware of the component-based structure of Android programs. This paper proposes a component-level exception mechanism for programmers to build robust Android programs with. With the mechanism, they can define an intra-component handler for each component to recover from exceptions, and they can propagate uncaught exceptions to caller component along the reverse of component activation flow. Theoretically, we have formalized an Android semantics with exceptions to prove the robustness property of the mechanism. In practice, we have implemented the mechanism with a domain-specific library that extends existing Android components. This lightweight approach does not demand the change of the Android platform. In our experiment with Android benchmark programs, the library is found to catch a number of runtime exceptions that would otherwise get the programs terminated abnormally. We also measure the overhead of using the library to show that it is very small. Our proposal is a new mechanism for defending Android programs from unexpected exceptions.

© 2015 Elsevier Ltd. All rights reserved.

1. Introduction

Exception handling in Java is an important feature to improve the robustness of Java programs. For example, Fig. 1 shows a simplified Java program that may throw one of two exceptions, `NoSuchOperator` and `ArithmeticException` (divide by zero), when users try to do a calculation with an unsupported operator or with zero as a divisor. Once the `calc` method throws such an exception, it is propagated along the call stack to a caller, the `main` method, where it is handled by the catch block.

Many Android programs are written in Java with Android APIs, presumably using exception handling. Android is Google’s open-source platform for mobile devices, and it provides the APIs (Application Programming Interfaces) necessary to develop applications for the platform in Java (http://developer.android.com). An Android program consists of components...
class Calculator {
    void main() {
        int a, b, r; char op;
        // read a, op, and b
        try {
            r = calc(a, op, b);
            // display the result
        }
        catch (NoSuchOperator e) {
            // Not support the operator
        }
    }
}

int calc(int a, char op, int b) {
    if (op=="+" ) return a+b;
    else if (op=="-" ) return a-b;
    else if (op=="*" ) return a*b;
    else if (op=="/" ) return a/b;
    else
        throw new NoSuchOperator();
}

Fig. 1. A Java program using exceptions.

such as activities, services, broadcast receivers and content providers. Android components communicate with another by sending messages called Intents. For example, an activity can start other activities by sending Intents to Android platform, which invokes methods of callee activities.

How vulnerable Android components are due to Intents have been reported by experiments in [1–4]. With Intent fuzzing, they generated random and semi-valid intents to test how components react to these exceptional conditions, focusing on uncaught exceptions that result in the crashes. One experiment by [2] measured the number of failed components for various types of components, reporting that 29(8.7%) out of total 332 activities crash with generated semi-valid intents. The distribution of exception types are also measured to understand how components fail due to uncaught exceptions, showing that NullPointerException makes up the largest share of all the exceptions, and other exceptions like ClassNotFoundException and IllegalArgumentException are next significant ones.

We have also found by examining source code of programs that Android programs can be very vulnerable to exceptions. We examined 9 programs and found that 41 activities (51%) out of total 80 activities have no exception handlers like try-catch, as will be shown in our experiment later. Activities without exception handlers cannot handle any thrown exceptions, and so result in the crashes when any exceptions are thrown.

From these observations, we can be sure that it is necessary for developing more robust Android programs to handle uncaught exceptions from components. Currently, programmers only resort to the conventional Java exceptions: the try-catch construct to defend statements and the thread-level uncaught exception handler interface (Thread.UncaughtExceptionHandler) to catch exceptions escaping from a thread. They are still crucial for Android programs, but they only address too fine-grained level in statements or too coarse-grained level in a thread. They are not immediately useful for addressing defending Android components.

Our design of component-level exception mechanism naturally follows that of the conventional Java exceptions of separating error-handling code from “regular” code and of propagating exceptions up the call stack. First of all, our mechanism is designed for each Android component to have a designated “catch” facility to defend itself from any (unexpected) uncaught exceptions thrown by the “regular” code of the component. This feature will allow programmers to focus more on the main flow of components, never missing any exceptions attempting to escape the components. Second, our mechanism is designed to support the propagation of exceptions following up a component activation stack. This facility will make components more resilient even by catching exceptions propagated from other components. Particularly, many components in Android programs have a relationship on “who activates whom”, which is very similar to a caller–callee relationship in method invocation. Also, Android platform already has an activity stack internally, which is the same as the call stack of method invocation, to maintain the who-activates-whom relationship.

In this paper, we propose a mechanism for component-level exception handling and propagation in Android programs, which can be used to make them more robust by defending themselves from unexpected events. We take a lightweight approach by providing new component APIs (e.g., ExceptionActivity class), which extends the existing Android components (e.g., Activity class) with the component-level exception mechanism. No Android platform needs to be modified to use our approach. Programmers can utilize component-level exception handling and propagation by writing components with the new extended APIs. This use of the exception mechanism preserves the structure of classes and methods in original programs. Our approach is also flexible in that programmers can take full control of deciding which components handle what exceptions and how they are recovered.

Following an overview of Android programs and our motivation in Section 2, we present our idea of the Android component-level exception mechanism in Section 3. We give a theoretical account on the mechanism by an Android semantics with exceptions to prove the robustness of the mechanism in Section 4. We also perform experiments in practice to show that Android programs can be more robust with the new API in Section 5. We count how many exceptions are caught with the new API. We also measure the marginal cost of the mechanism by changed lines of code, increased binary size, and startup time due to the adoption of the mechanism. Finally, after discussing related work in Section 6, we conclude in Section 7.
**2. Motivation**

An Android program is a Java program with APIs in Android platform. Using the APIs, one can build user interfaces to make a phone call, play a game, and so on. An Android program consists of components whose types are Activity, Service, Broadcast Receiver, or Content Provider. Activity is a foreground process equipped with windows such as buttons and text inputs. Service is responsible for background jobs, and so it has no user interface. Broadcast Receiver reacts to system-wide events such as notifying low power battery or SMS arrival. Content Provider supports various kinds of storage including database management systems.

Components in an Android program interact with each other by sending messages called Intent in Android platform. An Intent holds information about a target component to which it will be delivered, and it may hold data together. For example, a user interface screen provided by an activity changes to another by sending an Intent to Android platform, which will pause the UI screen and will launch a new screen displayed by a target activity specified in the Intent.

Fig. 2 illustrates a simplified Android calculator program. Activity class is a class that represents a screen in Android platform, and Main and Calc extending Activity are also classes representing screens. Initially, Android platform creates a Main object, it invokes the onCreate method to add three text input windows and one button with integer identifiers as arguments. After entering two integers and one operator, a user clicks the button. The new Intent speciﬁes the name of an activity class that represents the new screen and some data passed to the callee. The onClick method sets “Calc” and values from the text input windows in the new Intent object, and then it requests launching by invoking startActivityForResult(Intent i); Android accepts the request and changes the current UI screen by stacking Calc on Main, calling Calc’s onCreate method, which sets up a button to return the calculation result back to Main. On pressing the button, Android invokes Calc’s onClick method, which gets the operator and operands from the Intent (obtained by getIntent()), calculates it to set the result to the Intent, and dismisses Calc. Then Main appears again, and Android invokes the onActivityResult method to pass RESULT_OK as resultCode and the intent with the result as i, and to display it.

As well as the normal execution flow as explained above, the exceptional execution ﬂow might happen. When a user enters other than the four arithmetic operators, Calc’s onClick method will throw an exception, NoSuchOperator. When a user enters, say, “1 / 0”, the division in the method will throw ArithmeticException (divide-by-zero), which is an unexpected exception. In such exceptional cases, the exceptions thrown by the Calc’s onClick method will be propagated to Android platform who invoked the method, not to (the onClick method of) Main who activated Calc. Android platform then catches the exceptions, but it has nothing to do sensibly but stops the execution abnormally. Hence, the exceptions will never reach the try-catch block surrounding the invocation startActivityForResult(i) in the Main’s onClick method. This explains why the conventional exception handling is not effective for the Android component-based structure.

Currently, Android programs defend themselves against any exceptions only by the same ways as what plain Java programs do with. One is by a try-catch statement, and the other is by a thread-level uncaught exception handler (Thread.UncaughtExceptionHandler). The two conventional methods are still crucial for Android programs, but they are only

```java
class Main extends Activity {
    void onCreate() {
        addTextInput(1); // 1st operand
        addTextInput(2); // operator
        addTextInput(3); // 2nd operand
        addButton(1); // (=) button
    }

    void onClick(int button) {
        int a, b, op;
        Intent i = new Intent();
        i.setTarget("Calc");
        i.putExtra("op", op);
        i.putExtra("a", a);
        i.putExtra("b", b);
        try {
            int result;
            if (op == '+') result = a + b;
            else if (op == '-') result = a - b;
            else if (op == '*') result = a * b;
            else if (op == '/') result = a / b;
            Intent i = getIntent();
            i.putExtra("result", result);
            startActivity(i);
        } catch (Exception ex) {
            finish();
        }
    }
}

class Calc extends Activity {
    void onCreate() {
        addButton(1); // goback button
        // display the button
    }

    void onClick(int button) {
        int a, b, r;
        Intent i = getIntent();
        a = i.getIntExtra("a", 0);
        b = i.getIntExtra("b", 0);
        if (i.getStringExtra("op") == null) return;
        r = a + b;
        Intent i = new Intent();
        i.putExtra("result", r);
        startActivity(i);
    }
}
```

Fig. 2. An android calculator program with useless exception handling (in the onClick method of Main).
useful for defending too fine-grained level in statements or too coarse-grained level in a thread. They do not immediately address defending components, which are an important aspect of Android programs.

The current Java exception handling mechanism can make sense to Android developers, if they consider different components as loosely coupled ones like threads. As an Android program typically runs in a thread, making interleaved execution of several components, it is also possible for Android developers to consider components as less coarse-grained and more coupled ones than threads. From this view point, we can provide a mechanism to defend Android components against any uncaught exceptions as an alternative solution to the current Java exception handling mechanism.

The lessons from the conventional Java exception mechanism [5] can help us to design an alternative exception mechanism in terms of Android components. A notable advantage is that Java exceptions allow us to separate error-handling code from “regular” code cleanly. Similarly, it is possible for Android components to have their own “catch” block that enables programmers to write the main flow of the component code and to deal with the exceptional cases elsewhere. For example, exceptions like NullPointerException can be thrown by almost any pieces of codes in the component, but only one handler may be written in such a catch block of the component, rather than having a separate conditional statement or handler for each piece of the codes.

Also similarly as using Java exceptions allows us to propagate exceptions up the call stack automatically, it is possible for Android components to have an automatic mechanism on propagating exceptions up a component activation stack. For example, a callee Android component may propagate any exceptions to its caller component automatically when they are not handled by the callee. Clearly, this will make more chances to defend Android components from any uncaught exceptions thrown by their callee components. Without such a propagation mechanism, programmers may have to write to return error codes between callee and caller components, which will be very tedious and likely to make new errors particularly when an activation chain is long.

Now we have identified two things. First, to make Android programs robust, a new level of exception mechanism in terms of components exists in addition to the conventional Java exception mechanism. Second, to support a component-level exception mechanism, each component should provide a convenient way to catch any exceptions uncaught inside itself, which we call intra-component exception handling, and each caller component should also be able to handle any exceptions propagated from callee ones, which we call inter-component exception handling.

However, the current design of Android platform does not support such a component-level exception mechanism. In the example of Fig. 2, there is no component-level “catch” dedicated to each activity, and there is no convenient construct enabling the caller activity Main to catch exceptions propagated from the callee activity Calc. Later, we will formalize this problem by a semantics for Android activities and exceptions in Section 4.

This is a limitation of the design of Android platform solely depending on the Java exception semantics. To overcome the limitation, one might introduce a try-catch block surrounding the if statements in the Calc’s onClick method to handle both of the exceptions. This can be a very tedious work because we have to search all potentially vulnerable codes. We might not be able to find all such codes in advance. Even if we did do so, the main flow of a component would be mixed with the exceptional flow. Also, due to the separate roles of activities, one could not proceed further, for example, for Calc to get alternative inputs without going back to Main. This limitation motivates us to propose a component-level exception mechanism.

3. A component-level exception mechanism

We propose a new mechanism to provide a method named Catch to handle intra-component exceptions, and to propagate uncaught exceptions to its caller along the activation stack, which we call inter-component exceptions. Fig. 3 shows an example of the mechanism. Suppose Activity 1 starts Activity 2, which starts Activity 3. When an exception is thrown in Activity 3, it is passed to its Catch method. If it is not handled in the Catch method, then it is propagated to its caller Activity 2. If it is not handled in the Catch method of Activity 2, then it is propagated to Activity 1.

We design a domain-specific library for the component-level exception mechanism by extending Activity APIs, as in Fig. 4, which we call CE library. Basically, the library introduces new component classes (e.g., ExceptionActivity) by extending
the existing ones (e.g., Activity). Users of this library must write one’s own Activities by extending the new component class ExceptionActivity as in Fig. 5. The new component classes have a method named `Catch` as a default intra-component exception handler, which Android programs can override to define their own handler. For example, Fig. 4 shows ExceptionActivity’s `Catch` method that takes an exception as an argument and returns `false`, which means the exception is not handled. In Fig. 5, Calc class overrides the method to handle `ArithmeticException`. 
Inter-component try-catch construct: As well as having a single Catch method in each component to catch all intra-component exceptions, we can define an exception handler for each activation of components by TryActivityForResult(intent, catcher), a substitute method of startActivityForResult(intent) in the CE library, as follows:

```java
TryActivityForResult(intent, new Catch() {
    boolean handle(Throwable exn) {
        return true;
    }
    else return false;
}
});
```

where Catch is a Java interface in Fig. 4 for building a catcher object. The handle method returns true if the exception is processed. Otherwise it returns false to re-throw the exception.

Overriding Android interface methods: The CE library puts a fence at each border between each Android component and the platform to catch all exceptions from the component. For example, the onCreate method of ExceptionActivity in Fig. 4 invokes its counterpart method OnCreate, which is surrounded by a try-catch block (i.e., a fence) to catch all uncaught exceptions from the method and to propagate them to the caller activity. Android programs are supposed to override the counterpart method (OnCreate) to be called by the interface method (onCreate of ExceptionActivity), which Android platform invokes. Other interface methods such as onClick and onActivityResult have the similar fence structure.

For example, a calculator program in Fig. 5 is obtained by rewriting the library the previous Android example program in Fig. 2. Both Main and Calc classes now have counterpart methods OnCreate and OnClick instead of the original interface methods onCreate and onClick. Whenever any of the counterpart methods in Android programs misses catching any exception, the surrounding try-catch construct of the corresponding interface method in ExceptionActivity will catch it and propagate it by invoking “Throw(exn)” in the catch block of the interface method in ExceptionActivity.

Inter-component Throw construct: The CE library defines a (private) method “Throw(exn)” as shown in Fig. 4. It first passes an exception to the Catch method dedicated to intra-component exception handling. If the Catch method does not handle the exception, it then propagates the exception to a caller component via the standard Android return mechanism. For implementing this inter-component exception propagation, the library packages the exception into an Intent to set as a return value (“Intent i=new Intent(); i.setData(exn); finish(RESULT_EXN,i);”). Note that Intent class is assumed to have a field data, which holds an exception. In the Appendix (Fig. 8), a definition of Intent class is available. Also note that, to distinguish exceptional component results from normal ones, we tag the results with a result code, RESULT_EXN. Normal results are tagged with RESULT_OK. And then this component is dismissed (say, by invoking finish in ExceptionActivity). After the callee is finished with the intent holding the exception, Android platform will invoke the caller’s onActivityResult method to receive the intent.

On a caller’s receiving a result, the caller examines the result code in the onActivityResult method (of ExceptionActivity) to decide what to do next. If it is a normal result, an onActivityResult method is invoked. If it is an exception, the catcher (exception handler) of this activation will try to handle the exception. When the catcher succeeds in the exception handling, we get back to the normal state. Otherwise, we re-throw the exception by Throw method.

Note that there may be no more (say, ExceptionActivity extended) caller. In such a situation, the CE library is designed to stop inter-component exception propagation by finishing the last activity. One may opt to display a user a dialog window to notice the end of the exception propagation and the termination of this Android program.

To get the benefit from ExceptionActivity, we need to write an Android program with it or to rewrite an existing Android program into one using it. A basic transformation for a default recovery from any abnormal termination of an Android program is presented by Definition 1. For more fine-tuned recovery, a programmer should extend the basic transformation by overriding a Catch method for intra-component exception handling of an activity of one’s interest, or by installing an exception handler catcher for inter-component exception handling of an activity by TryActivityForResult(intent, catcher). Otherwise, by default, Catch is defined to return false and catcher is set to null, meaning not handling any exception but just propagating it.

**Definition 1 (A basic Android exception transformation).** For an Android program, we build a new one by applying the following rules:

- Every occurrence of Activity is replaced with ExceptionActivity.
- For each class declaration class C extends D where D is Activity or is inherited from it, the declared methods of onCreate, onClick, and onActivityResult are renamed as OnCreate, OnClick, and onActivityResult, respectively.
- Every occurrence of an invocation in the form of this.startActivityForResult(intent) is replaced with this.TryActivityForResult(intent,null) when this points to an object of Activity class or its descendant.
- The others remain unchanged.

Fig. 5 shows an example of how our component-level exception library is used to improve the robustness of the Android example program in Fig. 2. The library allows us to handle ArithmeticException thrown by the onClick method in Calc.
(when users enter, say, “1/0”) by the intra-component exception handler (the Catch method in Calc). Also, the library helps us to propagate NoSuchOperator exception (thrown by the same OnClick method when users enter other than the four arithmetic operators) to the inter-component exception handler (the handle method of a catcher created at the TryActivityForResult method invocation in Main). After the exceptions are handled, the program gets back to a normal state.

We have only described an exception mechanism extending Activity. The same idea can be applied to Fragment (which is a small detachable Activity). We also developed an exception mechanism for Service, Broadcast Receiver, and Content Provider as well. So, they are equipped with intra-component exception handling by the same idea of overriding interface methods of each component. These three types of components, however, have one’s own life-cycle, not always following the who-activates-whom relationship, hence we will use the inter-component exception propagation for the components only when domain knowledge on activation flows is available.

Besides the improvement of robustness, our proposal has a few advantages. First, our approach is lightweight, demanding no change on Android platform, and so all commercial Android devices can get benefit from it immediately. It even allows the mixed uses of Activity and ExceptionActivity. Second, using the CE library does not change the structure of the classes and methods at all, as shown by comparing the two programs in Figs. 2 and 5. Third, programmers can take full control of component-level exception handling and propagation by making use of Catch method of ExceptionActivity and Catch interface.

Using the CE library incurs only a little overhead as will be shown by our experiments later. The costs are threefold; efforts to rewrite programs with the library, increased binary sizes linked with it, and increased startup time due to the extra size and the exception handling layer. The experiment will show that our approach is very effective enough to catch a number of runtime exceptions but only with a little costs.

The proposed implementation works reasonably well with the component-based structure in Android programs. In the structure, for example, activities in a single program are loosely coupled by intents, and activities in two different programs can be seamlessly activated by one another as they are in the same program. Android components may be more tightly coupled with each other when any caller component has to know the kinds of exceptions thrown by its callee components. Programmers may need to have a trade-off in this respect. Nonetheless, such a tight coupling between components may be mitigated by catching an exception based on its group or general type rather than a very specific exception, according to the conventional grouping of Java exceptions by class hierarchy. For example, one can specify IOException to represent any type of error that can occur when performing I/O. Even when no knowledge on potential exceptions is available, one can specify Throwable, which is the root class of all Java exceptions. In addition, the usefulness of exception handling among components has been reported similarly by other researches [6–9], though they have dealt with server systems, which will be discussed later.

4. A formal semantics for Android-Java with exceptions and its theoretical properties

In this section, we give a theoretical account for the robustness property of Android programs using the component-level exception mechanism, which we explained by example in the previous section. We first present an Android semantics, which is based on an imperative version of the featherweight Java [10,11] extended with Java exceptions. The semantics supports the behavior of Activity in Android platform, such as the life-cycle of Activity and user interaction. This is expressive enough to run two Android examples in Figs. 2 and 5. After presenting the semantics, we prove the robustness property in this formal setting.

4.1. An android semantics with exceptions

The purpose of our semantics is to describe the execution of an Android program by a sequence of state transitions as $\text{state}_1 \Rightarrow \text{state}_2$ where states are Android program states. For a simple modeling, we identify four actions on Android platform to make a state transition as this: starting an Android program, activating an activity, user’s pressing some button, and finishing an activity to come back. Each state transition in the semantics exactly coincides to performing one of these actions. Each action $q$ on Android platform is defined as

$$ q ::= \text{Run} \; C \; | \; \text{Activate}(l) \; | \; \text{Press} \; btn \; | \; \text{Return}(c, l) $$

where $C$ is an activity class, $l$ is an (Intent) reference, $btn$ is an integer identifier for a button, and $(c, l)$ is an integer result code and a reference for a return value.

To describe an activity screen in the Android program, Android program states have the form of a triple $(t, q, h)$ where $t$ is an activity stack, $q$ is an action on Android platform, and $h$ is an object heap. An activity stack $t$ is $(l_1, w_1) \cdots (l_n, w_n)$ where $l_i$ is an activity reference and $w_i$ is a set of button windows in the activity. Only the top activity $(l_1, w_1)$ is visible to a user and the next top activity $(l_2, w_2)$ will be visible when we finish the top activity to remove from the stack. An action $q$ is one of what is described above or it can be empty as $\emptyset$. An object heap $h$ is a mapping of references onto objects as $(l_1 \rightarrow \text{obj}_1, \ldots, l_m \rightarrow \text{obj}_m)$ where $\text{obj} = \text{C}(f_1 = l_1, \ldots, f_m = l_m)$ with the fields $f$s and their values $l$s. An object may be written as $\text{C}(T = \emptyset)$ in shorthand.

We allow two different forms of states as follows: run $C$ for an initial state to start with an activity class $C$, and $\perp$ for the abnormal termination state due to some uncaught exception.
Fig. 6 defines a state transition relation for the four actions and for uncaught exceptions. An Android program runs as run $C \Rightarrow t_1, q_1, h_1 \Rightarrow \cdots \Rightarrow \text{state final}$ where either the program terminates normally with the empty activity stack (state final is $(\emptyset, \emptyset, h_{\text{final}})$) or it stops abnormally with an uncaught exception (state final is $\bot$).

Each state transition rule in Fig. 6 defines the execution of the corresponding action $q$ by the semantic function $A[q]$, which will be explained soon. This semantic function is a state transformer of the form

$$\lambda \text{state.}(\text{SuccOrExn, state'})$$

where SuccOrExn :: = Success $r$ | Exception exn to distinguish normal return values $r$ from exceptions exn. In (run), (launch), (button), and (back), the intended execution of each action is performed successfully while, in (exception), the execution of the action causes some uncaught exception.

The semantic function $A[-]$ for actions, which is assumed to be written in a call-by-value functional language like ML, is shown in Fig. 7. The semantic function takes an action $q$, and it changes states as the intended behavior of the action, resulting in a normal result or an exception. $A($Run $C)$ starts an Android program by creating an activity $C$, pushing it on the activity stack, and initializing it by invoking its interface method $\text{onCreate}()$. Another semantic function for Java expressions such as $\epsilon[x.onCreate()](x \mapsto l_{\text{new}})$ is extensively used in the semantic function for actions. We define $\epsilon[e]$ env as the Java semantics including exception constructs (throw and try-catch), where $e$ is a Java expression and env is an environment mapping identifiers to references. The details are available in the Appendix. $A[\text{Activate l}]$ for an Intent reference $l$ retrieves a target activity class $C$ from the intent reference and it launches the activity similarly as the steps for $A($Run $C)$ except assigning the intent reference $l$ to the intent field of the new activity. $A[\text{Press btn}]$ invokes the interface method $\text{onClick}(\cdot)$ of the top activity. $A[\text{Return}(c, h)]$ finishes the top activity and moves back to the second top activity if there is any one. On moving back, it invokes the interface method $\text{onActivityResult}(rc, rv)$ of the second top activity where $rc$ is an integer result code and $rv$ is a return value from the top activity. When there is no more activity on the stack except the top one, it stops the execution of an Android program with the activity stack empty.

It is easy to extend this semantic function further to support the full life-cycle of Activity by including the rest of the Activity interface methods such as $\text{onDestroy}$, $\text{onResume}$, and $\text{onPause}$, but we omit the extension due to the unnecessary complexity in our presentation.

$$
\begin{align*}
\text{(run)} & \quad A[\text{Run } C](t_1, q_1, h_1) = (\text{Success } r, (t, q, h)) \\
\text{(launch)} & \quad A[\text{Activate } l](t_1, q_1, h_1) = (\text{Success } (t', q', h'), (t, q, h)) \\
\text{(button)} & \quad A[\text{Press btn}](t_1, q_1, h_1) = (\text{Success } (t', q', h'), (t, q, h)) \\
\text{(back)} & \quad A[\text{Return}(c, l)](t_1, q_1, h_1) = (\text{Success } (t', q', h'), (t, q, h)) \\
\text{(exception)} & \quad A[q](t_1, q_1, h_1) = (\text{Exception } exn, (t', q', h')) \\
& \quad q \in \{\text{Run } C, \text{Activate } l, \text{Return}(c, l)\} \land q_0 = q \lor q \in \{\text{Press btn}\} \land q_0 = \emptyset
\end{align*}
$$

Fig. 6. State transition relations for actions on Android platform.

$$
\begin{align*}
A[\text{Run } C] & \quad = \text{do } l_{\text{new}} \leftarrow \epsilon[C x = \text{new } C(); x](\emptyset) \\
& \quad \quad \text{pushOntoActivityStack } (l_{\text{new}}, \emptyset) \\
& \quad \quad \epsilon[x.onCreate()](x \mapsto l_{\text{new}}) \\
A[\text{Activate } l] & \quad = \text{do } C \leftarrow \text{targetActivityClassFromIntent}(l) \\
& \quad \quad \text{do } l_{\text{new}} \leftarrow \epsilon[C x = \text{new } C(); x.(\text{intent} \mapsto \text{intent})](\emptyset) \\
& \quad \quad \text{pushOntoActivityStack } (l_{\text{new}}, \emptyset) \\
& \quad \quad \epsilon[x.onCreate()](x \mapsto l_{\text{new}}) \\
A[\text{Press btn}] & \quad = \text{do } l \leftarrow \text{getTopActivityRef} \\
& \quad \quad \epsilon[x.onClick()](x \mapsto l, h \mapsto \text{btn}) \\
A[\text{Return}(c, l)] & \quad = \text{do } l \leftarrow \text{getActivityStack} \\
& \quad \quad \text{if } \text{length}(l) \geq 2 \text{ then do} \\
& \quad \quad \quad \text{let } \{l_1, w_1\}, \{l_2, w_2\} : t_0 = l \\
& \quad \quad \quad \text{popFromActivityStack} \\
& \quad \quad \quad \epsilon[y.onActivityResult(rc, rv)](y \mapsto l_2, rc \mapsto c, rv \mapsto l) \\
& \quad \quad \text{else if } \text{length}(l) = 1 \text{ then do} \\
& \quad \quad \quad \text{popFromActivityStack} \\
& \quad \quad \text{else} \\
& \quad \quad \quad \epsilon[\text{throw new RuntimeException("EmptyActivityStack")}]() \emptyset
\end{align*}
$$

Fig. 7. Semantic functions for actions on Android platform.
Note that both of the semantic functions $\mathcal{A}[q]$ and $\mathcal{E}[e]$ env are written in monadic do notation to simplify passing states and checking whether a computation is performed successfully or not. For example, the definition of $\mathcal{A}[\text{Press btn}]$ is equivalent to one without do notation as

$$do \ \lambda \ state_0. \ bind \ \getTopActivityRef \ \lambda x.\ onClick\ (x\Rightarrow l, \ b\Rightarrow \text{btn}) \ = \ \lambda \ state_0. \ bind \ \getTopActivityRef \ \lambda x.\ onClick\ (x\Rightarrow l, \ b\Rightarrow \text{btn})$$

where $\text{bind} \ m_1(\lambda l.m_2)$ does case analysis on whether or not a computation $m_1$ is successful to decide to perform the next $m_2$. The full details on do notation and some auxiliary monadic functions such as $\text{pushOntoActivityStack}$ are also available in the Appendix.

Now we are ready to run, for example, two Android programs in Figs. 2 and 5. Let us first consider the Android program using Activity in Fig. 2. Assume a state that Calc activity is visible where the activity stack is the form of $\langle l_{\text{Calc}}, w_{\text{Calc}} \rangle. \langle l_{\text{Main}}, w_{\text{Main}} \rangle$. When a user presses a button $btn$ of an activity referenced by $l_{\text{Calc}}$, we invoke $\text{onClickListener}$ method of Calc by evaluating $\mathcal{E}[x.\ onClick\ (btn)] \ (x\Rightarrow l_{\text{Calc}}, \ b\Rightarrow \text{btn})$, according to $\mathcal{A}[\text{Press btn}]$. As explained in Section 2, this invocation may throw $\text{NoSuchOperator}$ exception when an illegal arithmetic operator is given. Throwing an exception is interpreted by the semantic function $\text{throw} \ exn = \lambda state. \ (\text{Exception} \ exn, \ state)$. In the case, this invocation evaluates to $(\text{Exception} \ l_{\text{exn}}, \ state)$, ignoring the execution of whatever follows the invocation, where $l_{\text{exn}}$ is an exception reference to $\text{NoSuchOperator} \ \{\ldots\}$ object. Consequently, we are forced to choose (exception) to make a state transition to $\bot$. In the alternative Android program using ExceptionActivity in Fig. 5, however, such a situation will never happen by propagating the exception ($l_{\text{exn}}$) to the second top activity Main, which will be shown in the following.

### 4.2. Robustness property

Every Android program using ExceptionActivity is more robust than the original program using Activity. We show this robustness property by proving that every Android program extending ExceptionActivity stops normally even when the original program extending Activity is terminated abnormally due to some uncaught exception thrown by itself.

Let us denote an Android program as $\mathcal{N}$, a set of class declarations where $N$ is a class declaration in the form of $\text{class } C \text{ extends } D \{\ldots\}$. $\mathcal{N}$ is an Android program rewritten using ExceptionActivity by the basic Android exception transformation in Definition 1. Then the robustness property is formulated as this theorem.

**Theorem 1** (Robustness). Suppose Android programs never try to start any activity that is absent. Whenever run $C\Rightarrow^n t, q, h$ or run $C\Rightarrow^n \bot$ in an Android program $\mathcal{N}$, run $C'\Rightarrow^{n+m} (t_{\text{final}}, q_{\text{final}}, h_{\text{final}})$ in the basic Android exception transformed program $\mathcal{N}'$ for some $n, m \geq 1$.

We prove the theorem by two propositions below. For the proof, we need to extend $(\sim)^*$ to states $(t, q, h)$ as $(t^*, q^*, h^*)$. $t^*$ and $q^*$ are simply $t$ and $q$, respectively. $h^*$ is the same as $h$ but every activity object is extended with an extra field of $\text{catcher}=\text{null}$ as follows: when $C$ is Activity or its descendant, $C[f=I, \text{catcher}=\text{null}]$ is replaced by $D[f=I, object] \text{such that } D$ is ExceptionActivity when $C$ is Activity or $D$ is the same as $C$ when $C$ is its descendant.

For the better recovery of exceptions, one could extend the basic transformation of Definition 1 by overriding a Catch method in an activity or by installing an exception handler $\text{catcher}$ in an activation of an activity. This will give rise to the better robustness than what one gets by the basic transformation.

First, the sound simulation proposition says that, whenever an Android program $\mathcal{N}$ arrives at a normal state, the transformed Android program $\mathcal{N}'$ also arrives at another state equivalent under $(\sim)^*$.  

**Proposition 1** (Sound simulation of normal execution). If run $C\Rightarrow^n t, q, h$ then run $C'\Rightarrow^{n}\sim t^*, q^*, h^*$ for $n \geq 1$.

Second, whenever an Android program $\mathcal{N}$ gets stuck throwing an exception, the transformed Android program $\mathcal{N}'$ will stop normally, propagating the exception to the last activity.

**Proposition 2** (Complete handling of exceptional execution). Suppose Android programs never try to start any activity that is absent. If run $C\Rightarrow^n \bot$ then run $C'\Rightarrow^{n+m} \emptyset, \emptyset, h$ for some heap $h$ and $n, m \geq 1$.

Note that, when an exception is thrown outside of the fence of the interface methods, our CE library (ExceptionActivity) will never be able to catch it. ActivityNotFoundException is one of such exceptions. This is thrown when one cannot find a target activity class to launch by targetActivityClassFromIntent($l$) from an intent reference $l$ in Fig. 7, due to the absence of the class.

The detailed proofs for the two propositions are available in the Appendix.

### 5. Experiments

Our benchmarks consist of nine Android source programs [13]: one commercial program (Bitcoin-Wallet), two sample programs (BluetoothChat and NotesList) developed by Google, one student project program (Cafe), and the rest five programs excerpted from advanced Android applications development books [12,14]. Table 1 shows a catalog of these source
programs. It also shows the numbers of Android components in each benchmark and the numbers (in the parentheses) of those using no try-catch blocks at all.

Our library is found to be very effective in catching a number of otherwise uncaught exceptions in runtime, as in Table 2. With our exception library, we have rewritten all benchmark programs, which are also available in [13]. In the experiment, we have found that the library catches 30 exceptions, which would otherwise lead the original benchmarks to a crash abruptly. Most of the exceptions caught by the library are unexpected ones by referencing NULL, using index out of bounds, and passing illegal arguments. Without the library, it would be difficult to catch the 30 exceptions. For example, NullPointerException could be thrown potentially many places in Android-Java programs, but adding all NULL checks would make the original source programs messy.

Table 2 also classifies the causes of the exceptions as malformed Intents, data/query format errors, mis-configurations, and so on. A malformed Intent may miss filling some field of the intent that the receiving component expects to have. Some benchmark improperly handles the format of XML data sent from a remote server due to the change of its XML schema. Also, some user-entered text may cause syntactic errors in query statements to build. Android platform versions of mobile devices may be different from what programs were developed with, which can also cause exceptions. The ratio of exceptions due to the malformed intents is higher than any others, which is consistent with the observation by [2].

We discuss how exceptions are thrown and how our library handles them in some of the benchmarks, BluetoothChat and NotesList. BluetoothChat is a text-based communication program using bluetooth. It consists of two activities and one broadcast receiver where the main activity launches the other discovery activity to find out any near bluetooth equipped devices to chat with. Once such a device is found, the broadcast receiver will receive an intent holding information about the device. NotesList is a memo program to create, to edit, and to delete memos in mobile database. It has three activities, one for listing memos, another for editing a memo, and the third for editing a memo title.

Both of them have been developed very robust by Google since the initial release of Android platform (2009), but it was not difficult to find out some vulnerability to cause exceptions. For example, we can construct a malformed intent for discovery of near-by bluetooth devices to be sent to the broadcast receiver of BluetoothChat to raise NullPointerException. Also, we can make NotesList raise the out-of-memory exception by copying a memo into itself repeatedly, that makes it large exponentially, terminating the memo editor activity with OutOfMemoryError. In both of the exceptional situations, the original Android programs terminate abnormally, but the rewritten ones catch them and they are able to return to the normal program state. We also found that these rewritten programs benefit from inter-component exception handling. This confirms our claim that the proposed component-level exception mechanism makes Android programs more robust.

The application of our Android exception library to our benchmarks involves rewriting them, which can be a criterion on how burdensome programmers are to use the library. Table 3 summarizes the changes of lines of code in the benchmarks. On average, we have changed 132 lines of code, which amounts to about 4.5% of the LOC of the original benchmarks. Compared with the LOC of each benchmark, the number is relatively small. It is straightforward to rewrite Android programs with the library under the basic Android exception transformation of Definition 1 with little domain knowledge on the Android programs.

The application of the exception library to our benchmarks also incurs some overhead. Table 3 shows increased binary sizes due to static linking with the library and increased startup time due to the extra size and the exception handling layer. In most of the benchmarks, the increased binary sizes range from 4K to 5K bytes, which do not take much storage at all even in mobile devices. Moreover, these increased binary sizes could be diminishing if Android platform supported a component-level exception mechanism as ours in a form of shared library.

Another overhead is the startup time of an Android program, defined as an interval between the time when user presses the launching icon and the time when the execution of the onCreate method of the main Activity finishes. The average startup time increases less than 0.03 s on Samsung Galaxy Nexus and Android Ver. 4.1.1. Even the longest difference (0.086 s in Bitcoin-Wallet) is never noticeable.

Note that, although we need to rewrite existing Android source programs with the CE library for comparison in the experiment, our intention is for programmers to start development with the library from the beginning. We believe that our

<table>
<thead>
<tr>
<th>Android apps</th>
<th>Activity</th>
<th>Service</th>
<th>Broadcast</th>
<th>Provider</th>
</tr>
</thead>
<tbody>
<tr>
<td>AndroidSecurity</td>
<td>11(6)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Bitcoin-Wallet</td>
<td>33(10)</td>
<td>2</td>
<td>5(2)</td>
<td>2</td>
</tr>
<tr>
<td>BluetoothChat</td>
<td>2(2)</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Cafe</td>
<td>5(4)</td>
<td>0</td>
<td>0</td>
<td>2(1)</td>
</tr>
<tr>
<td>Contacts</td>
<td>5(4)</td>
<td>1(1)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>MigrateClinic</td>
<td>6(4)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>NotesList</td>
<td>4(3)</td>
<td>0</td>
<td>3(1)</td>
<td>0</td>
</tr>
<tr>
<td>MediaPlayer</td>
<td>4(3)</td>
<td>0</td>
<td>3(2)</td>
<td>1(1)</td>
</tr>
<tr>
<td>Earthquake</td>
<td>10(5)</td>
<td>2</td>
<td>3(2)</td>
<td>1(1)</td>
</tr>
</tbody>
</table>
claims by the experiment will be valid for the intended course of development. The development of Android programs using
the component-level exception mechanism might lead to somewhat beneficial Android program architecture, which will be
an interesting future work.

6. Related work

Android applications can be vulnerable due to Intents, if input from Intents are not validated sufficiently. A malformed
Intent delivered to a receiver exposes attack surfaces as pointed out by [4]. For example, unauthorized receipt of an implicit
Intent can be made by malicious component, and Intent spoofing can be made, by which a malicious application sends an
Intent to an exported component that is not expecting Intents from that application. A static analysis tool like ComDroid
detects statically these potential vulnerabilities in Android applications [4].

It is also shown by experiments in [2] how vulnerable components are due to Intents. By extending the basic Intent
fuzzer [1], they generated random and semi-valid intents and tested how components handle these exceptional conditions.
In particular, they focused on uncaught exceptions, because they result in the crashes. In the experiment, they measured
the number of failed components for various types of components. For instance, 29(8.7%) out of total 332 Activities crash with
generated semi-valid intents on Android 4.0 emulator. The distribution of exception types are also measured to understand
how components fail due to uncaught exceptions. It is shown that NullPointerException makes up the largest share of
all the exceptions. In case of implicit Intents, the number of crashes due to NullPointerException is 32(38.5%) out of the
total 83 crashes. Other exceptions like ClassNotFoundException and IllegalArgumentException are next significant
ones. This experiment justifies a component-level exception mechanism to reduce abnormally failed components.

A combination of static analysis and random fuzzing was also proposed to dynamically test Android applications in [3]. A
path-insensitive, inter-procedural CFG analysis is employed to automatically extract the expected intent structure that a
component is expecting to receive. A set of intents is generated with the static intent structure information to explore more
execution paths. Target components are executed with these fuzzed intents, and both code coverage and crashes due to
exceptions are monitored.

An intent specification language was proposed by [15] for a common Intent fuzzing based Android testing framework,
providing a flexible way to express the shape information of intents and to generate test artifacts in various testing contexts.
In a preliminary result on an application of the testing framework, it reported in 10 real-world applications the inte-

Table 2
<table>
<thead>
<tr>
<th>Android Apps</th>
<th>Exns</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
</tr>
</thead>
<tbody>
<tr>
<td>AndroidSecurity</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Bitcoin-Wallet</td>
<td>7</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>BluetoothChat</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Cafe</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Contacts</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MigrateClinic</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>NotesList</td>
<td>11</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Earthquake</td>
<td>7</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Types of Exceptions

<table>
<thead>
<tr>
<th></th>
<th>Exns</th>
</tr>
</thead>
<tbody>
<tr>
<td>NullPointer</td>
<td>16</td>
</tr>
<tr>
<td>IndexOutOfBoundsException</td>
<td>3</td>
</tr>
<tr>
<td>ArrayIndexOutOfBoundsException</td>
<td>1</td>
</tr>
<tr>
<td>CursorIndexOutOfBoundsException</td>
<td>1</td>
</tr>
<tr>
<td>IllegalArgumentException</td>
<td>5</td>
</tr>
<tr>
<td>OutOfMemory</td>
<td>2</td>
</tr>
<tr>
<td>NumberFormat</td>
<td>1</td>
</tr>
<tr>
<td>IllegalState</td>
<td>1</td>
</tr>
</tbody>
</table>
Table 3
Lines of code, binary size in bytes, and startup time in millisecond of the benchmarks (with changes and increments when using the component-level exception library).

<table>
<thead>
<tr>
<th>Android apps</th>
<th>LOC</th>
<th>%</th>
<th>Binary size</th>
<th>%</th>
<th>Startup</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>AndroidSecurity</td>
<td>1297 (078)</td>
<td>6.01</td>
<td>323,070(+ 4001)</td>
<td>1.24</td>
<td>140( +08)</td>
<td>5.71</td>
</tr>
<tr>
<td>Bitcoin-Wallet</td>
<td>57,721 (592)</td>
<td>1.03</td>
<td>3,799,290(+ 13693)</td>
<td>0.36</td>
<td>1211(+ 86)</td>
<td>7.10</td>
</tr>
<tr>
<td>BluetoothChat</td>
<td>1124 (048)</td>
<td>4.27</td>
<td>27,087(+ 4837)</td>
<td>17.86</td>
<td>172(+ 15)</td>
<td>8.72</td>
</tr>
<tr>
<td>Cafe</td>
<td>2356 (110)</td>
<td>4.67</td>
<td>1,472,783(+ 5892)</td>
<td>0.40</td>
<td>321(+ 47)</td>
<td>14.64</td>
</tr>
<tr>
<td>Contacts</td>
<td>1781 (051)</td>
<td>2.86</td>
<td>35,663(+ 5801)</td>
<td>16.27</td>
<td>156(+ 08)</td>
<td>5.13</td>
</tr>
<tr>
<td>MigrateClinic</td>
<td>1111 (058)</td>
<td>5.22</td>
<td>53,545(+ 4916)</td>
<td>9.18</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>NotesList</td>
<td>3221 (088)</td>
<td>2.73</td>
<td>59,983(+ 5714)</td>
<td>8.63</td>
<td>312(+ 16)</td>
<td>5.13</td>
</tr>
<tr>
<td>MediaPlayer</td>
<td>664 (055)</td>
<td>8.28</td>
<td>153,833(+ 4586)</td>
<td>2.98</td>
<td>172(+ 16)</td>
<td>9.30</td>
</tr>
<tr>
<td>Earthquake</td>
<td>1934 (111)</td>
<td>5.74</td>
<td>50,676(+ 8736)</td>
<td>17.24</td>
<td>257(+ 16)</td>
<td>6.23</td>
</tr>
</tbody>
</table>

exception handling at architecture level. Contrary to this, our component-level exception mechanism is for the mobile platform, Android, in the form of an easy-to-use and user-extensible Java library using class inheritance to intercept exceptions systematically.

Another contribution of this paper is the semantic definition for combining Android, Java, and exceptions together. There have been researches on the semantics for each or two of the three features, but not for considering all of them. Also, it is noticeable that the notion of Monad is used for modeling exceptions and states in our Android-Java semantics, not in functional language semantics [21,22] where the technique of Monad is well known for structuring.

Starting with researches for Android semantics, Chaudhuri, Fuchs, and Foster [23,24] had presented an operational semantics for very abstract form of Android applications for the first time. They had used the semantics to prove the soundness of a type system for a permission-based security model and to formalize an information-flow analysis used in ScanDroid. Palamidessi and Ryan also defined another operational semantics for abstract Android security framework [25]. Payet and Spoto defined a non-standard operational semantics for a subset of Dalvik byte code instructions, particularly emphasizing the detailed life-cycle of Activity [26]. Jeon et al. took a step forward to define a precise operational semantics of Dalvik byte code instructions for dynamic analysis through symbolic execution in Symdroid [27]. Another operational semantics for Dalvik byte code instructions by Wognsen, Karlsen, and Olesen [28,29] had took into account the feature of exception handling at architecture level. Contrary to this, our component-level exception mechanism is for the mobile platform, Android, in the form of an easy-to-use and user-extensible Java library using class inheritance to intercept exceptions systematically.

As to researches for Java and exception semantics, Drossopoulou and Eisenbach firstly defined a formal semantics for Java [30]. Nipkow and Oheimb proposed a semantics named Java-light for a large subset of Java including exceptions to prove the Java type soundness [31]. Igarashi et al. proposed an operational semantics for core calculus of Java and Generic Java (GJ) [10]. Bierman et al. defined a semantics for an imperative core calculus of Java, employing an effect system to deal with the imperative features properly [32], which is similar to the use of Monad to model exceptions in our semantics. Jones [33] defined monadic functions for Java byte code instructions only over states, not exceptions. Stark et al. defined the most complete semantics for Java 1.0 using Abstract State Machines (ASMs), they also defined the compiler and the byte code format to prove the compiler correctness [34]. Another large-scale semantics for Java was defined by Farzan et al. using term rewriting in Maude [35]. This semantics is executable, and it was applied to model-checking of Java programs. Recently, a complete semantics of Java 1.4, called K-Java, has been presented by Bogdanaş and Roșu [36], and this work has been applied to model-checking multi-threaded programs.

7. Conclusion

We proposed a new idea of component-level exception mechanism for robust Android programs, and realized our idea as the flexible form of a domain-specific library. Theoretically, we have shown that the mechanism improves the robustness of Android programs by designing an Android semantics with exceptions. Also, in practice, six out of nine Android benchmark programs become more robust by using the library. In addition, the overhead of using the library in Android programs turns out to be small in terms of lines of code, binary sizes, and startup time, according to our experimental assessment.

Our proposal is a new Android program development methodology for recovering unexpected exceptions which is different from testing or static/dynamic/hybrid analyses [1-4,37]. As well as the prevention of abnormal termination, the proposal can also help us to prevent information exposure through any exception messages, as suggested by one of a set of software weaknesses (CWE-209, http://cwe.mitre.org).

In future, first of all, the component-level exception mechanism can guide a new design of Android platform architecture to enhance the current limitation of the platform. Second, our library can be extended to work with other foreign languages than Java. It only concerns Java-based Android programs now. One can mix Java with other programming languages such as JavaScript or C/C++ to build Android programs. A further research on how to propagate exceptions across codes written with JavaScript, being extensively used for cross-platform development, will be interesting.
Appendix A

We present the syntax and semantics of a featherweight Android-Java with exceptions, making complete the presentation of the semantics in Section 4. Our syntax and semantics start with those of the featherweight Java [10], but in an imperative version that makes such as heaps and reference updates explicit [11]. We extend it with the conventional Java exceptions, and support the behavior of Activity in Android platform, such as the life-cycle of Activity and user interaction.

Appendix A.1. A syntax of Android-Java with exceptions

We define a minimal syntax of a featherweight Android-Java, which allows us to write all examples in this paper, by extending the featherweight Java [10,11].

\[ N ::= \text{class} \ C \text{ extends} C \{ \text{Stmts} \} \]
\[ M ::= C \ m(C) \[ e \] \]
\[ e ::= x \ | \ xf \ | \text{new} \ C() \ | \text{if} \ e \text{ then } e \ | \text{else } e \ | \text{do } \text{Stmts} \]
\[ \text{let } x \text{ = } e; \text{ do } \text{Stmts} \]
\[ \text{try } e \text{ catch}(C) e \text{ throw } x \]

An Android program is a set of class declarations \( \mathcal{N} \). A block expression \( C \ x = e; e' \) declares a local binding of a variable \( x \) to the value of \( e \) for later uses in \( e' \). It is also used for sequencing \( e; e' \) by assuming omission of a dummy variable \( C \ x \). The conditional expression may be written as \text{if } e \text{ then } e \ | \text{else } e \ | \text{do } \text{Stmts} \]
\[ \text{let } x \text{ = } e; \text{ do } \text{Stmts} \]
\[ \text{try } e \text{ catch}(C) e \text{ throw } x \]

Fig. 8 shows our definition of Activity class and Intent class in the explained syntax. In the definition, the primitive functions \( \text{primStartActivity}, \text{primFinish}, \) and \( \text{primAddButton} \) are introduced in order to model the interaction between Android programs and platform. Their semantics will be defined in the next section.

Appendix A.2. A semantics for Android-Java with exceptions and proofs of its properties

Fig. 9 shows three sorts of basic semantic functions. Monadic functions (return and bind) make several semantics functions into a sequential one. Exception relevant functions (throw, trycatch) model how to throw exceptions and how to catch them in the semantics. State relevant functions (get and put) allow us to read and write the current state.

- Using the monadic functions bind and return, the do notation can be defined where \( \text{Stmts} \) is a sequence of \( x \leftarrow \text{exp} \), \( \text{exp} \), and \( \text{let } x \text{ = } \text{exp} \) separated by a semicolon, as follows:

\[
\begin{align*}
\text{do }\{x \leftarrow \text{exp}; \text{Stmts}\} &= \text{bind } \text{exp} (\lambda x. \text{do }\{\text{Stmts}\}) \\
\text{do }\{\text{exp}; \text{Stmts}\} &= \text{bind } \text{exp} (\lambda _. \text{do }\{\text{Stmts}\}) \\
\text{do }\{\text{exp}\} &= \text{bind } \text{exp} (\lambda x. \text{return } x) \\
\text{do }\{\text{let } x \leftarrow \text{exp}; \text{Stmts}\} &= \text{let } x \text{ = } \text{exp} \text{ in } \text{do }\{\text{Stmts}\}
\end{align*}
\]

- As variants of get and put functions, the four semantic functions (pushOntoActivityStack, popFromActivityStack, getActivityStack, and getTopActivityRef) used in Fig. 7 can be easily defined over the structure of states of the form \((t,q,h)\).

\begin{verbatim}
class Activity { 
  Intent intent;
  private void onCreate() { } 
  private void onPause() { } 
  private void onResume() { } 
  private void onDestroy() { } 
  private void onClick(int button) { 
    if (button==BACK) primFinish(resultCode, RESULT.CANCEL, null); 
    else { } 
  } 
  private void onActivityResult(int resultCode, Intent intent) { } 
  private void finish(int resultCode, Intent i) { primFinish(resultCode, i); } 
  private void tryActivityForResult(Intent i) { primStartActivity(i); } 
  private Intent getIntent() { this.intent; }
}

class Intent { 
  String target;
  Object data; // The setter and getter methods for the above fields
}
\end{verbatim}

Fig. 8. Android classes: Activity and Intent.
Two semantic functions throw and trycatch are introduced to support Java exception constructs later. throw exn state0 always returns (Exception exn, state0). trycatch m h state0 performs a computation by m state0. After that, we do case analysis on whether the computation is successful or not. When it succeeds, m state0 becomes the result of evaluation of trycatch m h, ignoring an exception handler h. When it throws an exception exn with state1, we give it to the handler h as h exn state1.

A function targetActivityClassFromIntent(l) is used in Fig. 7 to pick a target activity class from an Intent reference. Suppose h(l) = Intent{target = l1, …} for some heap h. The function returns Class(h(l1)) if l1 ≠ null where Class("C") = C such that C is an activity class. When the function fails to find any activity class due to the null Intent reference or the absence of any designated activity class, the function is defined to throw an exception by δ{throw new ActivityNotFoundException} envEmpty.

The semantic function δ[e] env for Java expressions e and environment env is defined in Fig. 10. The semantic function is a state transformer of the form

λstate.(SuccOrExn, state')

which is mostly standard [10]. The notable difference is to add an activity stack and an action to states for modeling Android platform. The standard Java constructs such as variable, field, and method invocation do not access nor change them while primitives change them as

- primFinish(x,y) replaces the current intent reference q with a new intent reference bound to x.
- primAddButton(x) dismisses the current activity to get back to its caller with a result code x and an intent y.
- primAddButton(x) adds a new button whose identifier is bound to x.

The semantic function uses some auxiliary functions defined in [10]. mbody(m,C) returns the body expression of the method of the class, and fields(C) gathers all fields belonging to the class, if necessary, following up the inheritance tree. D < C tests if D is any descendant class of C. The five semantic functions (getFromHeap, updateHeapWith, addToHeap, putAction, and addToWindows) used are also variants of the get and put functions.

The semantic function for Java expressions preserves the basic Android exception transformation and its extension over states, as proved by the following.

Lemma 1. If δ[e] env (t1, q1, h1) = (se, (t2, q2, h2)), then δ[e*] env (t1*, q1*, h1*) = (se, (t2*, q2*, h2*)) where se is either a normal result or an exception.

Proof. We prove this by induction on the depth of method invocation. For base cases, we can verify this proposition over the semantic functions for all kinds of expressions except x.m(y) in Fig. 10. For inductive case, the proposition holds for method invocation expressions by induction. Note that, every occurrence of z.startActivityForResult(intent) in e is replaced by z.startActivityForResult(intent, null) in e*. By the definition of the method TryActivityForResult (in Fig. 4), both of the method invocations do the same except that the latter one sets the catcher field of the activity z to null. The difference is identified by the definition of (−)∗ over heaps.

Now it is time to show the robustness theorem by proving the sound simulation of normal execution and the complete handling of exceptions as follows.

Proposition 1 (Sound simulation of normal execution). If run C⇒a t, q, h then run C′⇒a r, r', h* for n ≥ 1.

Proof. By the condition, every transition from run C is made by one of (run), (launch), (button), and (back), meaning that the corresponding application of the semantic functions A[Run], A[Activate], A[Press], or A[Return] leads to (Success result, state). This implies that all the sub-semantic functions such as δ[x.onCreate()] in A[Run] and A[Activate], δ[x.onClick(b)] in A[Press], and δ[x onActivityResult(rc, rv)] in A[Return] must evaluate to a successful result, too. By Lemma

bind mk = λstate. let (x, state') = m state in
case x of
  Exception e → (Exception e, state')
  Success r → k r state'
return r = λstate. (Success r, state)
get = λstate. (Success state, state)
put state0 = λstate. (Success (), state0)
throw e = λstate. (Exception e, state)
trycatch m h = λstate. let (x, state') = m state in
case x of
  Exception exn → h exn state'
  Success r → (x, state')

Fig. 9. Basic semantic functions.
1, the corresponding sub-semantic functions in the $(-)^*$—transformed Android program evaluate to the same successful result. By applying (run), (launch), (button), and (back) to the resulting state, we can get a successful transition by the transformed program, too.

**Proposition 2** (Complete handling of exceptional execution). Suppose Android programs never try to start any activity that is absent. If run $C \rightarrow^* n \bot$. then run $C \rightarrow^* n + m.0, h$ for some heap $h$ and $n, m \geq 1$.

**Proof.** By the condition of the proposition, there exists a transition with (exception) during the $n$ transitions from run $C$. This means that one of the semantic functions $A[Run], A[Activate], A[Press], \text{or } A[Return]$ with some state evaluates to (Exception exn, state). To have this kind of an exception that is uncaught by Android programs, there must exist some sub-semantic functions such as $E[x]: onCreate in A[Activate]$ that evaluates to the same exception and state. By Lemma 1 and by the definition of $onCreate$ in ExceptionActivity, in the $(-)^*$—transformed Android program, $x.OnCreate$ will be invoked, and $E[x.OnCreate]$ will evaluate to the same exception and state. In this situation, the try-catch block surrounding the invocation of $onCreate$ in $onCreate$ of ExceptionActivity will catch the exception and will pass it to the next top activity by $x.Throw(exn)$.

Note that every $(-)^*$—transformed program is defined to have only default component-level exception handlers, not explicitly made by programmers, as follows:

- The default Catch method of all (exception) activities is defined to return false.
- The default catcher field of all (exception) activities is set to null.

by the definition of ExceptionActivity class in Fig. 4.

Due to the definition of the default component-level exception handlers, any uncaught exception will be propagated across activities on the stack by repeating invoking $Throw$ and then $onActivityResult$ of ExceptionActivity until the activity stack becomes empty. During the propagation, the transformed Android program will create extra intent objects used for passing an exception to the previous activities by $Throw$ of ExceptionActivity. Therefore, $h$ is a union of $h_0$ and $(l_1 \rightarrow Intent\{\ldots\}, \ldots, l_k \rightarrow Intent\{\ldots\})$ where $h_0$ is a heap in state and $k$ is a positive integer.

In this case, it is easy to prove that the $(-)^*$ transformed Android program will terminate normally in a finite time after the original Android program abnormally terminates. This is because the length of the activity stack in state is finite. Therefore, there exists extra $m \geq 1$ transitions for the transformed Android program to take for the normal termination after $n$ transitions to keep pace with the original program. □
References


