A Semantics for Component-Level Exception Mechanism in Android

Kwanghoon Choi

Yonsei University, Wonju, Korea

Byeong-Mo Chang

Sookmyung Women's University, Seoul, Korea

Abstract

This paper proposes an Android semantics extended with a component-level exception mechanism.

Keywords:
Android, Java, Exception, Component, Semantics

1. Introduction

2. Overview of Android/Java

An Android program is a Java program with APIs in Android platform. Using the APIs, one can build mobile device 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 interact with each other by sending events called *Intent* in Android platform to form an application. The intent holds information about

Email addresses: `kwanghoon.choi@yonsei.ac.kr` (Kwanghoon Choi), `chang@sookmyung.ac.kr` (Byeong-Mo Chang)

```
class Main extends Activity {
   void onCreate() {
      addTextInput(1); // for 1st operand
      addTextInput(2); // for operator
      addTextInput(3); // for 2nd operand
      addButton(1); // for (=) button
      /* initialize the main screen */ }
   void onClick(int button) {
      int a, b;      char op;
      // read a,op,and b from the text input windows
      Intent i = new Intent();
      i.setTarget("Calc");
      // put a, op, and b into i
      i.setArg( ... a, op, b ... );
      try { startActivityForResult(i); } // Useless
      catch(NoSuchOperator exn) // Exception Handling
          { /* handle the exception */ } }
   void onActivityResult(int resultCode, Intent i) {
      if (resultCode == RESULT_OK) {
          // read r from i
          int r = ... i.getArg() ... ;
          // display the result
      } }
}
```

Figure 1: An Android Calculator Program with Useless Exception Handling (1/2)

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 destroy the UI screen and will launch a new screen displayed by a target activity specified in the intent.

To make Android programs robust, a caller activity should be able to handle exceptions thrown by callee activities. However, the design of Android program does not support such an inter-component exception handling. To illustrate this problem, Figure 1 and 2 presents an Android-based calculator program where the caller activity Main cannot catch exceptions from the callee activity Calc.

```
class Calc extends Activity {
   void onCreate() {
      addButton(1); // for return
      // display the return button
   }
   void onClick(int button) {
      int a, b, r;
      char op;
      Intent i = getIntent();
      // get a, op, b from i
      ... i.getArg() ... ;
      if (op == '+') r = a + b;
      else if (op == '-') r = a - b;
      else if (op == '*') r = a * b;
      else if (op == '/') r = a / b;
      else throw new NoSuchOperator();

      // put r into i
      i.setArg( ... r ... );
      // set i as a return value
      setResult(RESULT_OK, i);
      finish(); // dismiss this activity
   }
}
```

Figure 2: An Android Calculator Program with Useless Exception Handling (2/2)

Activity is a class that represents a screen in Android platform, and Main and Calc extending Activity are also classes representing a screen. Initially, Android platform creates a Main object, it invokes the *onCreate* method to add three input text windows and one button with integer identifiers as arguments. Now, a user can enter two integers and one operator to press the button, and then the *onClick* method is invoked to perform some action for the button. Intent is a class that represents an event to launch a new screen. It 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 read from the text input windows in the new intent object, and then it requests launching by invoking *startActivityForResult*. Android accepts the request and changes the current UI screen by stacking Calc on Main, calling Calc's *onCreate* method to setup a button for return of the calculation result back to Main. On pressing the button, Android invokes Calc's *onClick* method, which gets the operator/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 flows 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"*). 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 stop 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 exception handling is useless.

This is a limitation of the design of Android platform solely depending on the Java exception semantics. One might introduce a try/catch block surrounding the if statements in the Calc's *onClick* method to handle both of the exceptions, but could not proceed further to get alternative inputs without going back to Main.

To make more robust Android programs, we believe that Android platform should support an enhanced mechanism, which we call *inter-component exception handling*, to give caller activities more chance for handling exceptions from callee activities.

3. Inter-Component Exception Handling

Our goal is to redesign Activity APIs by incorporating our idea of inter-component exception handling so as to develop more robust Android programs. We design a domain-specific library for inter-component exception handling, as in Figure **??**. Basically, the library introduces new component classes (e.g., ExceptionActivity) inherited from the existing ones (e.g., Activity) to override the interface methods (e.g., onCreate) that Android plat-

```
class Main extends ExceptionActivity {
   void onCreate() {/* the same code as in Fig.2 */}
   void onClick(int button) {
      // the same code before the try/catch block
      tryActivityForResult(i, new Catch() {
         boolean handle(Throwable exn) {
            if (exn instanceof NoSuchOperator)
               { // handle the unsupported operator
                 return true; }
            return false;
         }  });
   }
   void onActivityResult(int resultCode,Intent i)
    {  // the same code    }
}
class Calc extends ExceptionActivity {
   void onCreate() { /* the same code */ }
   void onClick(int button) {/* the same code */}
   void Catch(Throwable exn) { /* a handler code */ }
}
```

Figure 3: An Android Program using Inter-Component Exception Handling

form invokes, by adding the inter-component exception handling capability. To use this library, Android programs must have the counterparts methods (e.g., OnCreate) rather than the original Android interface methods.

• Overriding Android Interface Methods: The inter-component exception handling (ICEH) library puts a fence at each border between an Android program and the platform to catch all exceptions from the program and to propagate them following the component activation stack. For example, the *onCreate* method of ExceptionActivity in Figure **??** invokes its counterpart method *OnCreate*, which is surrounded by a try/catch block to catch all uncaught exceptions from the method and to propagate them to some caller activity. The other interface methods to Android platform (e.g., *onClick* and *onActivityResult*) have the similar fence structure as this.

Figure 3 shows an Android calculator program using ExceptionActivity. For example, in the Main class, the original interface method *onCreate* is replaced with its counterpart method *OnCreate* to make use of ICEH library.

• Inter-component *Throw* Construct: To propagate an exception across components, the ICEH library offers "*Throw(exn)*", which packages the exception into an intent to set as a return value by invoking *setResult* with RESULT_EXN. The *onActivityResult* method of the caller will receive the intent together with the exceptional result code. Note that the standard Java construct "throw exn" can also be used to throw and propagate an exception within a component.

• Inter-component *Try/Catch* Construct: The ICEH library offers a caller component chances to handle exceptions thrown by callee activities as follows.

```
TryActivityForResult(i, new Catch() {
  boolean handle(Throwable exn) {
    if (exn instanceof E)
         { // handles the exception
           return true;  }
    else return false;
  }
});
```

where Catch is a Java interface for building a *catcher* object for inter-component exception handling. The *handle* method is designed to return true if the exception is processed. Otherwise it returns false and the exception will be re-thrown.

Instead of the code above, one could imagine the useless exception handling code as in Figure **??**. Because the *startActivityForResult* method behaves asynchronously, it immediately returns after requesting Android platform to launch an activity specified in the intent, and so no exception from the activity can be caught here.

Let us now consider inter-component exception propagation in detail. Suppose the *OnClick* method of Calc (extending ExceptionActivity) in Figure 3 throws a *NoSuchOperator* exception. The corresponding interface method *onClick* of Calc nets the exception by its try/catch block. Using "*Throw(exn)*", the exception is considered as a special activity result, and an intent with the exception is dispatched to the *onActivityResult* method of the caller activity (Main). To distinguish exceptional activity results from normal ones, the inter-component throw construct tags the results with a result code, RESULT_EXN. Normal results will be tagged with RESULT_OK. Note that Intent class is assumed to have a field *exn*, which holds an exception.

On a caller's receiving a result, the caller examines the result code in the *onActivityResult* method to decide what to do next. If it is a normal result, the method calls a (probably, user-defined) *OnActivityResult* method to return it. If it is an exception, we invoke the exception handling method *handle* of a *catcher*, which is once saved by *TryActivityForResult*. When the catcher succeeds in the exception handling, the current activity gets back to the normal state. Otherwise, we re-throw the exception either by propagating it to the next caller activity or by terminating the whole program abnormally when there is no more (ICEH-aware) caller. Note that Intent class is also assumed to have a field *calledByExnActivity* to indicate if there is more (ExceptionActivity extended) caller. The field is set by *TryActivityForResult*, and normal Android activities (extending Activity) leave it as NULL.

Therefore, in Figure 3, the ICEH library propagates each *NoSuchOperator* exception thrown by the *OnClick* method in Calc to the *handle* method of a catcher created at the *OnClick* method in Main. After the exception is processed, the program gets back to a normal state. This shows the robustness of the program in Figure 3.

This approach has a few advantages besides providing the robustness. It demands no change on Android platform, and so all commercial Android-based devices can get benefit from it immediately. It does not increase much the memory footprint of Android programs since they can share the same library. Developing new Android programs with the proposed library demands no more efforts than that without it. A disadvantage is that existing Android programs must be rewritten to use the library. However, the rewriting to adopt the library is straightforward and involves marginal modifications of the programs, as it is shown by the difference between the two Android programs with and without the library in Figure 3 and Figure **??**, respectively.

4. A Formal Semantics for Inter-Component Exception Handling

4.1. Syntax

The syntax of a featherweight Android-Java is defined by extending the featherweight Java [**?**].

$$
\begin{array}{lll}
N & ::= & \textsf{class}\ C\ \textsf{extends}\ C\ \{\bar{C}\ \bar{f};\ \bar{M}\} \\
M & ::= & C\ m(\bar{C}\ \bar{x})\ \{\ e\ \} \\
e & ::= & x \mid x.f \mid \textsf{new}\ C() \mid x.f = x \mid (C)x \mid x.m(\bar{x}) \\
 & \mid & \textsf{if}\ e\ \textsf{then}\ e\ \textsf{else}\ e \mid Cx = e;\ e \mid prim(\bar{x}) \\
 & \mid & \textsf{try}\ e\ \textsf{catch}(Cx)\ e \mid \textsf{throw}\ x
\end{array}
$$

A list of class declarations $\bar{N}$ denotes an Android program. 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 $\textsf{ite}\ e\ e\ e$ for brevity. We write a string object as a "string literal." Also, x.m("...") means $String\ s =$ "..."; $x.m(s)$ in shorthand. A recursive method offers a form of loops. The primitive functions $prim(\bar{x})$ are interfaces between an Android program and the Android platform, which will be explained later.

Using the syntax defined above, we can define a small set of Android class libraries in Figure 4 to model component-level activation flow in Android programs. In Activity, the member field (intent) will hold an intent object who activates this activity object. In Intent, the target field will be a target component to be activated, the data field will be an extra argument to the target component, and the action field will describe a service that any activity activated by this intent will provide. For notation, the empty method body { } intends to return nothing, denoted by *void*, and may be written as {*void*}.

4.2. Semantics

Now we turn to the semantics part. Figure 5 shows basic semantics functions, which will be used for Android/Java semantics later. We borrow the idea of Monad to structure in a modular way the Android/Java semantics with the notions of exceptions and states, resulting in the introduction of the following two artifacts.

First, we distinguish exceptions from normal return values as

$$Excetional\ ::=\ Success\ r\ \mid\ Exception\ e$$

Second, we regard the meaning of each Java expression as a state transition function of the form $\lambda state.(r, state')$ that takes an initial state $state$, and returns a result r changing the state to $state'$.

Combining the two artifacts, we model the semantic functions of Java expressions in the form of

$$\lambda state.\ (Exceptional,\ state')$$

```
class Activity {
    Intent intent;
    Catcher catcher;

    void onCreate() { }
    void onPause() { }
    void onResume() { }
    void onDestroy() { }
    void onClick(int button) {
       if (button==BACK) primFinish(RESULT_CANCEL, null);
       else {}
    }
    void onActivityResult(int resultCode, Intent intent) { }
    void addButton(int button) { primAddButton(button); }
    void tryActivityForResult(Intent i, Catch catcher) {
       if (catcher==null)
          this.catcher = new Catch();
       else
          this.catcher = catcher;
       primStartActivity(i);
    }
    boolean Catch(Throwable exn) { return false; }
    Intent getIntent() { this.intent; }
}
class Intent {
    String target;
    Object data;
    String action;
    // The setter and getter methods
    // for the above fields
}
class Catch {
    boolean handle(Throwable exn) { return false; }
}
```

Figure 4: Android Classes: Activity and Intent

$$\begin{array}{lcl}
\textit{return } r & = & \lambda state.\ (Success\ r, state) \\
\textit{bind } m\ k & = & \lambda state.\ let\ (x, state') = m\ state\ in \\
 & & \quad case\ x\ of \\
 & & \qquad Exception\ e \rightarrow (Exception\ e, state') \\
 & & \qquad Success\ r \rightarrow k\ r\ state' \\
\textit{throw } e & = & \lambda state.\ (Exception\ e, state) \\
\textit{trycatch } h\ m & = & \lambda state.\ let\ (x, state') = m\ state\ in \\
 & & \quad case\ x\ of \\
 & & \qquad Exception\ e \rightarrow h\ e\ state' \\
 & & \qquad Success\ r \rightarrow (Success\ r, state') \\
\textit{get} & = & \lambda state.\ (Success\ state, state) \\
\textit{put } state_0 & = & \lambda state.\ (Success\ (), state_0)
\end{array}$$

Figure 5: Basic Semantic Functions

using the basic semantic functions in Figure 5.

The basic semantic functions are threefold. Monadic functions (*return* and *bind*) are used for combining arbitrary semantics functions into one. Exception relevant functions (*throw*, *trycatch*) models how to throw exceptions and how to catch them in the semantics. State relevant functions (*get* and *put*) help to read and write the current state.

For readability of semantics functions, we also borrow the do notation for Monad as

$$do\ \{\ Stmt_1\ \cdots\ Stmt_n\ exp\ \}$$

where

$$Stmt\ ::=\ exp\ ;\ \ |\ \ x \leftarrow exp\ ;\ \ |\ \ let\ x = exp\ ;$$

Using the basic monadic functions, we can define the do notation as follows.

$$\begin{array}{lcl}
do\ \{\ exp\ \} & = & exp \\
do\ \{\ x \leftarrow exp;\ Stmts\ \} & = & bind\ exp\ (\lambda x.\ do\ \{\ Stmts\ \}) \\
do\ \{\ exp;\ Stmts\ \} & = & bind\ exp\ (\lambda\_.\ do\ \{\ Stmts\ \}) \\
do\ \{\ let\ x = exp;\ Stmts\ \} & = & let\ x = exp\ in\ do\ \{\ Stmts\ \}
\end{array}$$

As a formal model of Android programs, we define an operational semantics for the featherweight Android/Java. A triple (t, q, h) is intended to form the configuration of a screen in an Android program. An activity stack t is

$$\text{(run)} \quad \frac{\mathcal{C}[\![Run]\!]\ C\ (\emptyset,\emptyset,\emptyset) = (Success\ r, (t,q,h))}{run\ C \Longrightarrow t,q,h}$$

$$\text{(launch)} \quad \frac{\mathcal{C}[\![Activate]\!]\ l(t, Activate\ l, h) = (Success\ r, (t',q',h'))}{t, Activate\ l, h \Longrightarrow t',q',h'}$$

$$\text{(button)} \quad \frac{\begin{array}{l}\text{Button } btn \text{ is pressed}\\ \mathcal{C}[\![Press]\!]\ btn\ (t, Press, h) = (Success\ r, (t',q',h'))\end{array}}{t, Press, h \Longrightarrow t',q',h'}$$

$$\text{(back)} \quad \frac{\mathcal{C}[\![GoBack]\!]\ c\ l\ (t, GoBackWith(c,l), h) = (Success\ r, (t',q',h'))}{t, GoBackWith(c,l), h \Longrightarrow t',q',h'}$$

$$\text{(exception)} \quad \frac{\mathcal{C}[\![q]\!]\ \cdots\ (t,q,h) = (Exception\ e, (t',q',h'))}{t,q,h \Longrightarrow \bot}$$

Figure 6: Semantic Rules for Android Platform

$(l_1, w_1) \cdot \cdots \cdot (l_n, w_n)$ where w denotes a set of button windows in an activity. A command of activity transitions q is defined as,

$$q\ ::=\ Press \mid Activate(l) \mid GoBackWith(c,l)$$

where we abbreviate each of q as $\emptyset$, l, and (c,l), respectively. An object heap h is in the form of $\{l_1 \mapsto obj_1, \cdots, l_n \mapsto obj_n\}$ where $obj = C\{\bar{f} = \bar{l}\}$.

We write an object of class C as $C\{\bar{f} = \bar{l}\}$ with the fields $\bar{f}$ and their values $\bar{l}$. For example, $Intent\{target = l, data = l', action = l''\}$ denotes an intent object. l is a String reference for the name of a target component, l' is another object as an argument, and l'' is another String reference for an action description. Following the convention, an object may be denoted by its reference.

$t, q, h \Longrightarrow t', q', h'$ denotes an activation flow between the two top activity components l_1 and l_1', which is activated by the intent q. $\bar{l}$ and $\bar{l}'$ may be the same. $\Longrightarrow^*$ denotes zero or more steps.

Note that each new activity reference piles up on the stack in the order of activation. Only the top activity l_1 is visible to a user and the next activity l_2 becomes visible when the top activity is removed.

```
$\mathcal{C}[\![Run]\!] = \lambda C.\ FENCE\ ($
        $do\ l_0 \leftarrow [\![C\ x = \mathsf{new}\ C(); x]\!]\ \emptyset$
           $l_1 \leftarrow [\![x.OnCreate();\ x.OnResume()]\!]\ \{x \mapsto l_0\}$
        $)$
$\mathcal{C}[\![Activate]\!] = \lambda l.\ FENCE($
        $do\ h \leftarrow getHeap$
           $let\ C = target(l, h)$
           $t \leftarrow getActivityStack$
           $let\ (l_1, w_1) \cdot t_0 = t$
           $[\![z.OnPause()]\!]\ \{z \mapsto l_1\}$
           $l_2 \leftarrow [\![C\ x = \mathsf{new}\ C();\ x.intent = intent;\ x]\!]\ \{intent \mapsto l\}$
           $l_3 \leftarrow [\![x.OnCreate();\ x.OnResume();\ x]\!]\ \{x \mapsto l_2\}$
        $)$
$\mathcal{C}[\![Press]\!] = \lambda btn.\ FENCE($
        $do\ t \leftarrow getActivityStack$
           $let\ (l, w) \cdot t_0 = t$
           $l_3 \leftarrow [\![x.OnClick(b)]\!]\ \{x \mapsto l, b \mapsto btn\}$
        $)$
$\mathcal{C}[\![GoBack]\!] = \lambda c, \lambda l.\ FENCE($
        $do\ t \leftarrow getActivityStack$
           $if\ length(t) \geq 2\ then\ do$
              $let\ (l_1, w_1) \cdot (l_2, w_2) \cdot t_0 = t$
              $[\![x.OnPause()]\!]\ \{x \mapsto l_1\}$
              $[\![y.OnActivityResult(rc, rv);\ y.OnResume()]\!]\ \{y \mapsto l_2, rc \mapsto c, rv \mapsto l\}$
              $[\![x.OnDestroy()]\!]\ \{x \mapsto l_1\}$
           $else\ if\ length(t) == 1\ then\ do$
              $let\ (l, w) = t$
              $[\![x.OnPause();\ x.OnDestroy()]\!]\ \{x \mapsto l\}$
           $else\ [\![throw\ new\ RuntimeException(\text{“}EmptyActivityStack\text{”})]\!]\ \emptyset$
        $)$
```

Figure 7: Semantic Functions for Android Platform

$$
\begin{array}{lcl}
[\![x]\!]\ env & = & do\ \ return\ env(x) \\
[\![x.f_i]\!]\ env & = & do\ \ h \leftarrow getHeap \\
 & & \qquad let\ l_x = env(x) \\
 & & \qquad let\ C\{\bar{f} = \bar{l}\} = h(l_x) \\
 & & \qquad return\ l_i \\
[\![x.f_i = y]\!]\ env & = & do\ \ h \leftarrow get \\
 & & \qquad let\ l_x, l_y = env(x), env(y) \\
 & & \qquad let\ C\{\bar{f} = \bar{l}\} = h(l_x) \\
 & & \qquad putHeap\ h\{l_x \mapsto C\{\bar{f} = \bar{l}_{1,i-1} l_y \bar{l}_{i+1,n}\}
\end{array}
$$

... To be filled soon ...

$$
\begin{array}{lcl}
[\![throw\ x]\!]\ env & = & do\ \ throw\ env(x) \\
[\![try\ e_1\ catch(C\ x)\ e_2]\!]\ env & = & do\ \ trycatch\ ([\![e_1]\!]\ env)\ (\lambda l. \\
 & & \qquad if\ l\ instanceof\ C \\
 & & \qquad then\ [\![e_2]\!]\ env\{x \mapsto l\} \\
 & & \qquad else\ throw\ l)
\end{array}
$$

Figure 8: Semantic Functions for Java Expressions

A set w of button windows is merely a set of integer identifiers for buttons appearing on the screen being displayed. This is the minimal machinery to allow users to interact with Android programs. A heap h is a mapping of references into objects.

Android platform allows each intent to specify a target activity either explicitly by giving a target class name or implicitly by suggesting only actions. The former is called explicit intents, useful for the intra-application components, and the latter is called implicit intents, useful for the inter-application components [1]. Our Android semantics models both of explicit and implicit intents.

To pick a target activity class from an explicit/implicit intent reference, we define a function $\mathsf{target}(l, h)$ as: Suppose $h(l) = Intent\{target = l_t, action = l_a, ...\}$, and then the function returns

- $Class(h(l_t))$ if $l_t \neq null$
- $IntentFilter(h(l_a))$ if $l_t = null$ and $l_a \neq null$

where

$$\begin{array}{lcl}
[\![primStartActivity(x)]\!]\ env & = & do\ \ putCmd\ (Activate\ env(x)) \\
[\![primAddButton(x)]\!]\ env & = & do\ \ w \leftarrow getWindows \\
 & & putWindows\ (w \cup \{env(x)\}) \\
[\![primFinish(x, y)]\!]\ env & = & do\ \ putCmd\ (GoBackWith\ (env(x), env(y)))
\end{array}$$

Figure 9: Semantic Functions for Primitive Expressions

- $Class(\text{“C”}) = C$ such that C is an activity class, and
- $IntentFilter(\text{“}action_i\text{”}) = C_i$, a mapping table of actions (strings describing services) onto activity classes that are capable of supporting the services.

When the $\mathsf{target}(l, h)$ fails to find any activity class, it is defined to return activity-not-found error.

Every Android program accompanies a manifesto file declaring various kinds of properties of the classes including such intent filters. In this paper, we assume such a manifesto file exists simply in the form of *IntentFilter* function for our purpose.

In Figure 6, our Android platform is modeled in the form of non-deterministic semantic rules that depends on many monadic semantic functions of the form $(\lambda state.(Exceptional, state'))$ explained before. In the semantic functions, states are triples in Android platform. Hence the semantics functions will have the refined form as

$$\lambda(t, q, h).\ (Exceptional, (t', q', h'))$$

Recall that *get* and *set* functions to read and write states in the semantic functions. We can extend these functions to *getActivityStack*, *setActivityStack*, *getCmd*, *setCmd*, *getHeap* and *setHeap* to read and write the corresponding elements of the states in a similar way.

Also recall that a task stack t is a list of activities $(l_1, w_1) \cdot \cdots \cdot (l_n, w_n)$. We can define *popActivityStack* as a semantic function to remove the top activity from the activity stack. We can also extend the state read and write functions to $getWindows$ and $setWindows$ to read the window set of the top activity and to write it.

The semantic rules in Figure 6 define a single step of Android platform in terms of Activity. They heavily depends on the semantic functions for activity transition commands defined in Figure 7.These semantic functions all have the form of $\lambda(t,q,h).\ (Exceptional,(t',q',h'))$. The current triple of Android platform is given to the semantic functions as an argument, and the result triple of the functions will be the next triple as in (run), (launch), (button), and (back) if we get a normal result as *Success r*. Otherwise, we get into abnormal termination, denoted by $\perp$, due to uncaught exception, as shown in (exception).

The semantic functions for activity transition commands in Figure 7 all have the form of

$$\lambda x_1.\cdots\lambda x_n.\ trycatch\ (\ some\_action\ )\ interCompHandler$$

where "some_action" models the behavior of Android platform for a given activity transition command. Suppose we have a command *Activate l* to activate an Activity with an Intent *l*. Android platform will invoke the *OnPause* method of the top activity, create a new Activity (designated by *l*), and invoke the *OnCreate* and *OnResume* methods of the new activity in sequence, as the semantic function $\mathcal{C}[\![Activate]\!]$ is defined. The other semantics functions for the commands *Press* and *GoBack* can be similarly read.

Note that the modeling action is surrounded by two semantic functions, *trycatch* with *interCompHandler*. This is for handling exceptions uncaught inside activity. The semantic function *interCompHandler* models inter-component exception propagation and handling along the activity stack as in the SPE paper. This will be explained soon.

The semantic functions for Java expressions are defined in Figure 8. The two constructs *throw* and $try\ e_1\ catch(C\ x)\ e_2$ can be directly implemented with the two semantic functions *throw* and *trycatch*, which we defined before.

For primitive expressions for interacting with Android platform, Figure 9 defines another set of semantic functions. *primStartActivity(x)* replaces the current intent reference q with a new intent reference bound to x. *primAddButton(x)* adds a new button whose identifier is bound to x. *primFinish(x,y)* dismisses the current activity to go back to its caller with a result code x and and an intent y.

Finally, Figure 10 defines two semantic functions for inter-component exception propagation. First, *interCompHandler* tries to handle an uncaught exception inside an activity by the *Catch* method of the activity. When it succeeds, it gets back to a normal state. Otherwise, it invokes the second

```
$interCompHandler\ =$
        $\lambda e.\ do\ t \leftarrow getActivityStack$
            $if\ length(t) == 0\ then\ halt\ exn$
            $else\ do\ let\ (l_1, w_1) \cdot t_0 = t$
                    $trycatch\ (do\ b \leftarrow [\![z.Catch(exn)]\!]\ \{z \mapsto l_1, exn \mapsto e\}$
                              $if\ b = true\ then\ halt\ exn$
                              $else\ interCompHandler'\ e$
                    $)\ (\lambda e.\ interCompHandler'\ e)$

$interCompHandler'\ =$
        $\lambda e.\ do\ popActivityStack$
            $t \leftarrow getActivityStack$
            $if\ length(t) == 0\ then\ halt\ exn$
            $else\ do\ let\ (l_1, w_1) \cdot t_0 = t$
                    $trycatch\ (do\ b \leftarrow [\![Catch\ c = x.catcher;\ c.handle(exn)]\!]\ \{x \mapsto l_1, exn \mapsto e\}$
                              $if\ b = true\ then\ halt\ exn$
                              $else\ \ interCompHandler\ e$
                    $)\ (\lambda e.\ interCompHandler\ e)$

$halt\ =\ \lambda exn.\ return\ void$
```

Figure 10: Semantic Functions for Inter-Component Exception Propagation

semantic function *interCompHandler*$'$ which pops the top activity away and moves to the second top (or caller) activity. This second semantic function tries to handle the exception by (the *handle* method of) a catcher in the caller activity. The semantics functions repeat this procedure recursively until either the exception is properly processed or there is no more activity.

5. Properties of the Android Exception Semantics

Let us define an Android semantics $\Longrightarrow_f$ as $\Longrightarrow$ where every occurrence of $FENCE$ in C[-] is replaced with a specified function f. For example,

- $\Longrightarrow_{ID}$ denotes an Android semantics where ID is an identity function.
- $\Longrightarrow_{EXN}$ where $EXN = trycatch\ interCompHandler$, denotes another Android semantics supported by the component-level exception handling mechanism.

The following simulation proposition says that, whenever an Android program under the conventional Android semantics ($\Longrightarrow_{ID}$) arrives at a normal state, the Android program also arrives at the same normal state under the exception enhanced Android semantics ($\Longrightarrow_{EXN}$).

Proposition 1 (Sound Simulation of Normal Execution)**.** *If run $C \Longrightarrow^n_{ID}$ t, q, h then run $C \Longrightarrow^n_{EXN} t, q, h$ for $n \geq 1$.*

Proof. The condition implies that the evaluation of $\mathcal{C}[\![\,]\!]$ for each intermediate (t_i, q_i, h_i) leads to $(Success\ r, (t_{i+1}, q_{i+1}, h_{i+1}))$ in the conventional semantics.

By the definition of $trycatch\ interCompHandler\ m$ where $m\ (t_i, q_i, h_i)$ is $(Success\ r, (t_{i+1}, q_{i+1}, h_{i+1}))$. After the case analysis on x (which is $Success\ r$), we will get $(Success\ r, (t_{i+1}, q_{i+1}, h_{i+1}))$, which is the same as in the conventional semantics. □

What will happen if an Android programs throws an exception under the conventional semantics? The Android program under the exception-enhanced semantics will lead to a normal state. This observation will be formalized by the following proposition.

Proposition 2 (Complete Handling of Exceptional Execution)**.** *If run $C \Longrightarrow^*_{ID}$ $\perp$ then either there exists (t, q, h) such that run $C \Longrightarrow^*_{EXN} t, q, h$ or it loops infinitely.*

Proof. The condition of evaluating to $\perp$ implies that $(\mathcal{C}[\![q]\!]\ \cdots\ (t_0, q_0, h_0))$ becomes

$$(Exception\ e, (t', q', h'))$$

under the conventional semantics where the identity fence bypasses the exception e. In the exception-enhanced semantics, however, the term $(\mathcal{C}[\![q]\!]\ \cdots)$ is surrounded by the exception fence function $trycatch\ interCompHandler$, which will take it for the first argument m. The state (t_0, q_0, h_0) will be the seoncd argument $state$ of the exception fence function.

After applying m to $state$ inside the exception fence function, we do a case analysis on x. Since x is $Exception\ e$ as we see above, we follow the corresponding case branch and evaluate $h\ e\ state'$, which is

$$interCompHandler\ e\ (t', q', h')$$

Now we need to prove that the inter-component exception handler above will always β-reduce to some term of the form $(Success\ r, (t, q, h))$, meaning that there is no uncaught exceptions. We prove this by induction on the length of t'.

Case $\mathit{legnth}(t') = 0$: The definition of $\mathit{interCompHandler}$ will choose the "then"-branch of if on the length of activity stack, and so it will have reductions as follows:

$$\begin{array}{ll} = & \mathit{interCompHandler}\ e\ (t', q', h') \\ \rightarrow_{\beta}^{*} & \mathit{halt}\ e\ (t', q', h') \\ \rightarrow_{\beta}^{*} & (\mathit{Success}\ \mathit{void}, (t', q', h')) \end{array}$$

Assuming that the proposition to prove holds for all activity stacks of length n, we prove the case of length $n + 1$.

Case $\mathit{legnth}(t') = n+1$: The definition of $\mathit{interCompHandler}$ will choose the "else"-branch of if on the length of activity stack. Now we have three sub cases. The first sub case is "*halt exn*", and the second and third sub cases are "$\mathit{interCompHandler}'\ e$".

We safely assume that "$[\![z.\mathit{Catch}((\mathit{exn})]\!]\ \{z \mapsto l_1, \mathit{exn} \mapsto e\}\ (t', q', h')$" terminates. Otherwise, the proposition is simply true.

When the reduction arrives at the first sub case, the proposition simply holds by the definition of *halt*. Otherwise, we need to consider the chum function $\mathit{interCompHandler}'$ under the condition of $\mathit{legnth}(t') = n + 1$.

In the application of $\mathit{interCompHandler}'$ to e, the first monadic action is "*popActivityStack*", which will decrease the length of activity stack by one. The definition of $\mathit{interCompHandler}'$ will then select either the "then"-branch or "else"-branch of if on the length of activity stack.

The sub case of "then"-branch satisfies the proposition by the similar argument as for $\mathit{interCompHandler}$.

The further reduction after choosing "else"-branch will lead to three sub cases. Again we can assume that "$[\![\mathit{Catch}\ c = x.\mathit{catcher};\ c.\mathit{handle}(\mathit{exn})]\!]\ \{x \mapsto l_1, \mathit{exn} \mapsto e\}$" terminates due to the presence of the potential infinite loops in the conclusion.

The proposition holds in the first sub case trivially. It also holds in the second and third sub cases by induction hypothesis since the length of activity stack is descreased by one. $\square$

In the proof above, one may claim that it might not be legitimate to use the induction on the length of activity stack when considering the fact that any invocation of "$[\![z.\mathit{Catch}(\mathit{exn})]\!]$" or "$[\![c.\mathit{handle}(\mathit{exn})]\!]$" could start an activity and so could increase the length of activity stack. However, such an invocation can only request to launch an activity. It can never increase the length of activity stack inside the transition of $\Longrightarrow$. Therefore, the use of induction on the length of activity stack is applicable in the proof.

Now we state a robustness theorem.

Theorem 1 (Robustness)**.** *Every Android program under the exception enhanced Android semantics is more robust than the same one under the conventional Android semantics.*

Proof. The theorem follows the two propositions above. When every conventional Android program terminates normally, the exception-enhanced program behaves the same, by Proposition 1.

When every conventional Android program throws some uncaught exception terminating abnormally, the exception-enhanced program either terminates normally (by handling the exception properly) or it gets into some infinite loop (during the exception handling), by Proposition 2. $\square$

6. Related Work

A survey of proposals on Android Semantics is as follows. (Java-level or Dalvik-level)

- An operational sematics for abstract Android applications used in ScanDroid for information-flow analysis [8, 9]. Another semantics for abstract Android security framework [10]
- An operational semantics of Dalvik for extending a path-based termination analysis to Android programs [11]
- An operational semantics of Dalvik bytecode used in Symdroid for dynamic analysis through symbolic execution[12]
- An operational sematnics of Dalvik bytecode with reflection and the WebView JavaScript interface [13, 14]
  - This work defines several semantic rules for$java.lang.Class.forName$, $java.lang.Class.getMethod$, $java.lang.Class.newInstance$, $java.lang.reflect.Method.$ methods.
  - It also defines semantic rules for the JavaScript interface such as $android.webkit.WebView.addJavaScriptInterface$.

- A featherweight semantics for Android/Java [7]

A survey of proposals on Java and Exception semantics is as follows.

- Semantics for core calculus of Java and GJ [15], and for an imperative Java with effects [16]

7. Conclusion

We have proposed an Android semantics extended with a component-level exception mechanism.

[1] http://developers.android.com

[2] G. Huang and Y. Wu, Towards Architecture-Level Middleware-Enabled Exception Handling of Component-based Systems, the 14th Int'l ACM SIGSOFT Symp. on Component-Based Software Engineering, Colorado, USA, June 21-23, 2011.

[3] A. Romanovsky, Exception Handling in Component-based System Development, 25th Annual Intl' Computer Software and Applications Conference, pp.580-586, 2001.

[4] F. C. Filho, P. A. C. Guerra, V. A. Pagano and C. M. F. Rubira, "A Systematic Approach for Structuring Exception Handling in Robust Component-based Software", J. Braz. Comp. Soc., Vol.10, No.3, pp.5-19, 2005.

[5] C. Dellarocas, Toward Exception Handling Infrastructures in Component-based Software, International Workshop on Component-based Software Engineering, Vol.31, 1998.

[6] Kwanghoon Choi and Byeong-Mo Chang, "A Type and Effect System for Activation Flow of Components in Android Programs", Information Processing Letters, 114(11):620-627, November 2014.

[7] Kwanghoon Choi and Byeong-Mo Chang, "A Type and Effect System for Activation Flow of Components in Android Programs", Information Processing Letters, 114(11):620-627, November 2014.

[8] A. Chaudhuri, Language-based Security on Android, Proceedings of ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, Dublin, Ireland, June 15, 2009.

[9] A. P. Fuchs, A. Chaudhuri, J. S. Foster, Scandroid: Automated Security Certification of Android Applications, Tech.Rep. Technical Report CS-TR-4991, Dept. of Computer Science, University of Maryland, 2009.

[10] Catuscia Palamidessi and Mark D. Ryan, Formal Modeling and Reasoning about the Android Security Framework, Trustworthy Global Computing, Lecture Notes in Computer Science, Vol.8191, pages 64-81, 2013.

[11] Étienne Payet and Fausto Spoto, An Operational Semantics for Android Activities, In Proceedings of ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation, San Diego, California, USA, January 20-21, 2014.

[12] J. Jeon, K. K. Micinski, and J. S. Foster, SymDroid: Symbolic Execution for Dalvik Bytecode, Technical Report CS-TR-5022, Department of Computer Science, University of Mayrland, College Park, July 2012.

[13] Erik Ramsgaard Wognsen and Henrik Søndberg Karlsen, Static Analysis of Dalvik Bytecode and Reflection in Android, Master Thesis, Software Engineering, Aalborg University, June 6, 2012.

[14] Henrik Søndberg karsen, Erik Ramsgaard Wognsen, Mads Chr. Olesen, and René Rydhof Hansen, Study, Formalisation, and Analysis of Dalvik Bytecode, Science of Computer Programming, Vol.92, pages 25-55, 2014.

[15] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler, Featherweight Java: A Minimal Core Calculs for Java and GJ, ACM Transactions on Programming Languages and Systems, pages 132-146, 1999.

[16] G. M. Bierman and M. J. Parkinson, A. M. Pitts, MJ: An Imperative Core Calculus for Java and Java with Effects, Technical Report, University of Cambridge, 2003.