# Providing A Basis For Verifying Android Applications In JML

Alexander Drechsel
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
a.w.b.drechsel@student.utwente.nl

## ABSTRACT
Android applications are often hard and time consuming to verify. This is in part due to the fact that currently available tools and specification languages are not sufficient for the verification of Android applications. Validation of applications, however, is important to ensure reliability and correctness. This paper provides a basis for the development of a Java Modelling Language extension to assist in specifying and verifying Android applications. Java Modelling Language is a specification language which tools can use to verify Java code. The specification and verification works by annotating every method with requirement assertions which should be true when the method is called and assertions which the method body should ensure become true after a method has been executed. This paper proposes supporting Android validation by developing a JML extension designed for Android code and provides a basis for such an extension.

## Keywords
Android, Java Modeling Language, Verification, Specification

## 1. INTRODUCTION
Android devices are being sold in increasing numbers and the Android OS is achieving a larger market share of smartphones each year[4]. With this increase in sales the number of applications developed for Android increases each year as well. The quality of these developed applications varies greatly, with some being error-prone to a large degree.

Though the Android API prevents such error-prone applications from harming the overall system, such applications may, for example, be unusable or unnecessarily consume large amounts of resources. One of the factors contributing to this variance in quality and reliability is the fact that Android applications are difficult to properly validate. By validation we mean ensuring that an application is correct by formally checking that it satisfies its own requirements. This is because of several reasons:

- Android applications are multithreaded. This is the main source of difficulty because most formal specification languages and their related tools are not yet able to handle this properly.
- Not all of an application's behaviour is written in Java code, some is specified in xml-files.
- An application may be interrupted by other parts of itself or different application but they must be able to restore their state afterwards.


*21th Twente Student Conference on IT*, June 23st, 2014, Enschede, The Netherlands.


These are the reasons why existing tools and languages for validation are not sufficient to properly validate Android applications and most validation must be done by hand. This way of validation, however, is time consuming and does not guarantee that the application does not react in ways the developer did not foresee.

One of the languages which can be used for specification and validation is Java Modelling Language[6] (JML). JML is a specification language for Java which works via Design by Contract. Design by Contract is a programming method where each method is given a contract in an annotation. This contract indicates that a method will ensure certain conditions to be true after the method is executed provided certain requirements hold. JML can be used to specify Android code because Android code is based on Java. However, Android, due to its architecture, poses new challenges which make Android applications difficult to handle with currently existing JML techniques.

This paper proposes supporting Android validation by developing a JML extension designed for Android code and provides a basis for such an extension. We achieve this by briefly discussing the API and what should be done related to that, we study Activity transitions and provide specification clauses to quickly describe their behaviour. By specification clause we mean in this paper a semantically specific part of a formal specification indicated by a keyword. We also provide a specification clause for a group of functions which support specification of concurrent behaviour. Finally, we discus Android permissions and how to verify that the correct and only the correct permissions have been requested. This paper addresses parts of the multithreaded aspects and behaviour outside of Java code but does not address interruptions by other applications or the Android OS.

## 2. BACKGROUND
In the following we provide background information about JML and Android that is necessary for this paper.

### 2.1 Java Modeling Language
JML[6] or Java Modeling Language is a specification language for Java. JML specification follows the Design by Contract[8] paradigm using Hoare logic[5]. Hoare logic is a formal system with a set of logical rules for reasoning about the correctness of computer programs. Hoare logic works using Hoare triplets which is a set of assertions with a command. If the first

assertion holds true then after execution of the command the second assertion will hold true.[5].

Logic assertions are added to Java code as Java annotations for each method and variable.

A simple example of JML specification is given in Figure 1. This example is about a simple counter which has methods to increase or decrease the counter and a method to return the current count. The count may be not negative which is asserted using a invariant. Both the method to increase the counter and to return the current count do not have a requirement which means in Hoare logic that their first assertion is simply true. The method to decrease the counter, however may potentially lower the count to negative values. To prevent this from happening the requirement `number>=amount` was added to assert that no number larger than the current count is subtracted. The `ensures` clause in each method forms the second assertion and in this simple example exactly describes the effect of each method.

After specification has been added the validity is verified, either manually or using verification tools. There are various verification tools which use JML but they can be broadly divided into two categories: Static Checkers and Runtime Checkers.

Static checkers verify applications by using its source code and the associated specification without executing any code.

Runtime Checkers verify applications by executing the application and verifying for each state if the associated specification holds.

### *2.1.1 Ghost variables*

Ghost variables are specification only variables which function similarly to Java variables. A ghost variable may be created anywhere a Java variables may be created plus in method headers. Values can be assigned to ghost variables using the JML set command anywhere a Java variable could have a value assigned. A ghost variable will then have this value in the same manner as a Java variable. Like Java variables, ghost variables have a scope and if declared inside a method body will only exists within that body.

### *2.1.2 Final notes about JML*

JML is still in development and has many variants and extensions. One weakness of JML and Hoare logic in general is that it handles concurrency poorly due to the fact that sequential execution of methods cannot be guaranteed.

```
//@invariant number>=0;
private /*@spec_public@*/ int number;

//@assignable number;
//@ensures number == \old(number)+ amount;
public void addToTotal(int amount){
        number = number+Math.abs(amount);
}

//@requires number>=amount;
//@assignable number;
//@ensures number == \old(number)+ amount;
public void substractFromTotalNoNegative(int
amount){
        number = number-Math abs(amount);
}

//@ensures \result == number
Public /*@ pure @*/ int getNumber(){
        return number;
}
```

**Figure 1. A simple example of a JML specification.**

## 2.2 Android

Android is a software platform for mobile devices that includes an operating system, middleware and key applications. [7] Applications for Android are usually created in Java with the help of the Android SDK and design guidelines. Android applications are designed using four components:

- `Activities` which are used for user interfaces and are the core of each application usually starting other components,
- `Services` which perform actions in the background for an unlimited period of time,
- `Broadcast Receivers` which simply react to announcements,
- `Content Providers` which allow applications to make parts of the applications data available to other applications.[7]

### *2.2.1 Activities and the activity lifecycle*

Activities are the most used components in Android and are managed using the Activity Lifecycle where activities are either *alive*, *paused*, *stopped* or *finishing*. It is important to note here that stopped activities, though hidden, still contain all its state and member information[7]. This is important because the application can return to this activity and thus must remain valid. Finally it should be noted that most interaction between user and application occurs via events which happen concurrently.

#### *2.2.1.1 The Activity class*

In code `Activites` all possess the same general framework. An activity has a `onCreate` method which is called to create the Activity. The `onCreate` method broadly functions the same way as a constructor for a Java class, initializing the various elements in the class. After the `onCreate` method is finished and the `Activity` is brought to the foreground the `onStart` method is called which should be used to initialize or restore volatile memory consuming data or actions. `onStart` is also called when a *stopped* `Activity` is restored. The `onBackPressed` method is a method which is called when the user presses the back button on his or her Android device. By default this method simply ends this method and restarts the

previous `Activity` indirectly calling its `onStart` method but may be overridden to have various other functions. Aside from these methods an `Activty` will have one or more methods which will be called when a button on the screen is pressed. These methods then handle the user interaction and are generally the source of transitions to other activities.

Activities which are more complex will need to override more methods from the `Activity` class such as the `onStop` method but the information given above is sufficient for the examples given in this paper.

In Figure 2 is an example of a basic `Activity`. Though not strictly necessary the `onStart` and `onBackPressed` methods have been explicitly added even though they simply call their parent method.

```
public class InstructionActivity extends Activity
{

        @TargetApi(Build.VERSION_CODES.ICE_CREAM_
SANDWICH)
@Override
protected void onCreate(Bundle
savedInstanceState) {
        super.onCreate(savedInstanceState);
        this.setRequestedOrientation(ActivityInfo
.SCREEN_ORIENTATION_LANDSCAPE);
        overridePendingTransition(R.anim.fadein,
R.anim.fadeout);
        setContentView(R.layout.activity_instruct
ion);
        getActionBar().setDisplayHomeAsUpEnabled(
false);
        if(Build.VERSION.SDK_INT>=Build.VERSION_C
ODES.ICE_CREAM_SANDWICH)
        getActionBar().setHomeButtonEnabled(false
);
        }

protected void onStart(){
                super.onStart();
        }

public void onBackPressed(){
                super.onBackPressed();
        }
}
```

**Figure 2. Example of a basic `Activity`.**

### *2.2.2 Properties defined in xml-files*

Not all of the properties and behaviour of Android Activities are contained within their Java class, some are contained within the associated XML file and the Application's AndroidManifest.xml file.

The AndroidManifest.xml file is the core file of an Android application and presents all essential information of the application. Among this information is a description of all Activities. This description gives the capabilities of each Activity and its launch behaviour.

All Activities have an associated xml-file which has the same name as their Java class and describes the initial state of the GUI for that activity and the names for all its elements.

### *2.2.3 Permissions in Android*

Android applications by default are not allowed to access the full functionality of Android and can only access functions which according to the android developer guide "cannot do anything that would adversely impact the user experience or any data on the device."[1]. Any application which requires access to protected function will need to set the required permission for the protected functions in its AndroidManifest. When installing an application Android will request the user to grant an application its required permissions.

## 3. CONTRIBUTION

This paper provides a basis for a JML extension for specifying and validating Android applications. This basis is provided by examining several basic components nearly every Android application will have.

The research for this paper was mainly done using a case study. The case study is an Android version of the Pong video game and was originally developed for OVSoftware as a student project[2]. The case study is a relatively simple application having only Activities and non-Android classes. During specification of the case study a class named Annotation was added to the application to serve as a container for several global variables which were added for specification purposes. While the extra class was not strictly necessary, it improves readability compared to adding the variables an existing class because they are not strictly part of the application.

From this case study we have drawn several conclusions which contribute to a JML extension for specifying and validating Android applications. We identified a way to specify transitions from one activity to the next using ghost variables and propose a short way of writing this. Motivated by the splash screen and its common use in Android applications we identified what constructs are necessary to specify a set of concurrency methods from Android of which `postDelayed` is one example.

Separate from this case study, we also researched permissions and identified a clause which can be used to verify an Android application requests the permissions it needs and does not request unneeded permissions.

## 4. PRELIMINARIES: THE API

A side problem for any specification of code that refers to APIs or Libraries is that typically no openly available formal specification exists for these libraries. This specification is important because it defines the framing of each method. By framing we mean what variables and objects a method modifies. Framing is specified in JML using `pure` and `assignable`. This framing is important because it is a main method of reasoning for static checkers and greatly boosts execution speed when runtime checking. Because of the size of the Android API and the ubiquity of its use within Android application code, it is especially important that some form of formal specification of the Android API exists when specifying Android applications.

Methods which change no variables or objects are specified are `pure`. The only methods which may be used directly in specification expressions are methods which are `pure`. Methods which do modify variables and objects should have an `assignable` clause. The assignable clause should specify which variables and objects are changed.

Static checkers require that framing is specified for each method. They require this framing to reason about variables and comparing the specification to the source code.

Though it is not necessarily required for runtime checkers it is generally desired for framing to exist because it greatly boosts execution speed. This increase in execution speed is because instead of needing to recheck the complete state space it only

needs to recheck a smaller portion of the state space. It should be noted that this increase in execution speed requires that the assignables are indeed correct. With runtime checking it is possible to verify that the assignables are correct though this is difficult.

For this research we will assume that it is already known which methods are pure and that for non-pure methods at least some form of framing has been specified, such that they can be used.

Initial estimates of which methods are pure can be done by assuming that all "get" and "is" methods are pure. While this should cover most methods which are pure within the Android API it could have false positives, where a get method aside from returning the requested data also increases a counter which tracks the number of times the data has been requested or a similar variable change, and false negatives, where a method which does not use the naming convention is pure. An example of a false negative we encountered during the case study is `UptimeMillis()` from the `android.os.SystemClock` class whose informal specification suggests that it is a pure method.

Specification of which variables non-pure methods modify is somewhat more involved but can be abstracted by encompassing the variables and objects a method changes inside an abstract frame. One way of abstracting the modified variables and objects into a frame is by using dynamic frames as explained in Dynamic Frames in Java Dynamic Logic by Schmitt et al[12].

Of course such estimated and abstracted specifications would need to be checked against the actual implementation of the Android API but that is not part of this work and such verification would be quite difficult[9][13]. For the purposes of this paper estimated specifications using the assumptions mentioned above were generally sufficient.

# 5. ACTIVITIES

In this section we discuss how Activity transitions in Android can be specified with JML. While these transitions can be described without an extension to JML, this is quite lengthy and has much duplication. To this end we propose two new JML specification clauses: `activityTransition` and `activity`. We discuss the intuition behind these two new clauses here and a final version of its syntax and semantics is given in Appendix A.

## 5.1 Activity transitions

One of the aspects which makes Android difficult to specify in standard JML is the transitions from one Activity to the next. Not only is this difficult because Android is a concurrent platform but the Android environment can change applications by interrupting them with time-sensitive events such as a phone call or by destroying stopped Activities to free up memory.

An Activity does not directly transition to another Activity but instead creates an intent which is sent to a handler which will set up and execute the actual transition. For our case study we referenced Midlet Navigation Graphs in JML by W. Mostowski and E. Poll[10] on how we could describe screen transitions and like in [10] mainly used ghost variables to describe each transition, abstracting much of the complexity.

## 5.2 Specifying using basic JML

During the specification of the case study we used ghost variables and the abstraction they provide to specify activity transitions. For each transition we specified what the next Activity should be and to which Activity it should return to if the back button is pressed.

Ghost variables alone, however, do not necessarily reflect the actual state of an application, thus we expanded the receiving `onStart()` method using `hasWindowFocus()` to verify that the receiving Activity is *alive* and on the foreground. We also made it a requirement that intents for Activity transitions should only be created by `Activities` for which `hasWindowFocus()` is true. This is generally not required by Android. But we added it because typically only the active Activity should trigger an activity transition. The final version of this specification clause should also allow this to not always be a requirement. Additionally, we added a new ghost variable `screenTransition` to indicate the volatile state the application enters when transitioning to a new Activity (that is, an intermediate state between the active activities). Finally, we did not concern ourselves with framing during this part of the case study. Figures 3 and 4 have been taken from this part of the case study.

```
/* @requires Annotation.screenTransition==false;
@requires this.hasWindowFocus();
@ensures Annotation.next ==
Annotation.LOBBYACTIVITY;
@ensures Annotation.previous ==
Annotation.MAINMENUACTIVITY;
@ensures Annotation.screenTransition==true;
@ensures quickplaybutton.isEnabled()==true; */
public void customPlay(View view) {
if(!quickplaybutton.isEnabled()){
        quickplaybutton.setText(R.string.quickpla
y_button);
        quickplaybutton.setEnabled(true);
        NetworkService.getInstance(this).update(n
ull, new QuickPlayEvent());
        }
/* @set Annotation.screenTransition=true;
@set Annotation.next=Annotation.LOBBYACTIVITY;
@set Annotation.previous =
Annotation.MAINMENUACTIVITY; */
Intent intent = new Intent(this,
LobbyActivity.class);
NetworkService.getInstance(this).update(null, new
JoinLobbyEvent());;
startActivity(intent);
}
```

**Figure 3. Method from `MainMenuActivity` from the case study initiating a transition to `LobbyActivity` specified with basic JML.**

```
/* @requires Annotation.screenTransition==true;
@requires Annotation.next ==
Annotation.LOBBYACTIVTY;
@requires Annotation.previous ==
Annotation.MAINMENUACTIVITY;
*/
protected void onStart(){
//@set Annotation.screenTransition=false;
super.onStart();
}

/* @requires Annotation.screenTransition==false;
@requires this.hasWindowFocus();
@ensures Annotation.next ==
Annotation.MAINMENUACTIVITY
@ensures Annotation.previous ==
Annotation.LOBBYACTIVITY;
@ensures Annotation.screenTransition==true; */
public void onBackPressed(){
/*
@set Annotation.screenTransition=true;
@set Annotation.next =
Annotation.MAINMENUACTIVITY;
@set Annotation.previous =
Annotation.LOBBYACTIVITY;
*/
super.onBackPressed();
}
```

**Figure 4. Relevant methods from `LobbyActivity` for receiving transitions specified with basic JML.**

## 5.3 Specifying using extended JML

While manually specifying each transition in this manner is possible, this results in lengthy and duplicate specification for `Activity`, as Figures 3 and 4 clearly show.

To alleviate this we propose to introduce a few new semantic clauses to function as shorthands ("An abbreviated symbolic writing method that increases speed and brevity of writing"[14]) for Activity transitions.

Methods which initiate a transition should be specified with the help of `activityTransition(String targetActivityName)`. An example of `activityTransition` is given in Figure 5. Using the information given in this clause, a tool utilising this proposed extension can expand the clause to the required specification. An example of this expansion is given in Figure 6.

The shorthand for the receiving side of the transition is more complex because it spans multiple methods, namely `onCreate`, `onStart` and `onBackPressed` are all affected by how an application can be started and its place in the activity history stack. Because multiple methods are affected by this shorthand, we decided that a logical location for the semantic clause specifying this would be in the header of the class so that the Activity as a whole can be specified. The clause we propose for the receiving side is: `activity(String[] launchingActivities, String[] BackingActivities, String[] backableActvities)`

Each of the variables in this clause provides information about a different part of its transition behaviour:

- `launchingActivites`: The names of the `Activities` which can launch this `Activity`.
- `backingActivities`: The names of the `Activities`, which can transition to this `Activity` when the back button is pressed.
- `backableActivities`: The names of the `Activities`, this `Activity` can transition to when the back button is pressed.

An example of this clause being used is given in Figure 7 and the way this clause is expanded in shown in Figure 8.

```
/* @activityTransition("LobbyActivity")
@ensures quickplaybutton.isEnabled()==true;
*/
public void customPlay(View view) {
if(!quickplaybutton.isEnabled()){
        quickplaybutton.setText(R.string.quickpla
y_button);
        quickplaybutton.setEnabled(true);
        NetworkService.getInstance(this).update(n
ull, new QuickPlayEvent());
        }
Intent intent = new Intent(this,
LobbyActivity.class);
NetworkService.getInstance(this).update(null, new
JoinLobbyEvent());;
startActivity(intent);
}
```

**Figure 5. Example use of the `activityTransition` specification clause.**

```
/* @requires Annotation.screenTransition==false;
@requires this.hasWindowFocus();
@ensures Annotation.next ==
Annotation.LOBBYACTIVITY;
@ensures Annotation.previous ==
Annotation.MAINMENUACTIVITY;
@ensures Annotation.screenTransition==true;
@ensures quickplaybutton.isEnabled()==true;
*/
public void customPlay(View view) {
if(!quickplaybutton.isEnabled()){
        quickplaybutton.setText(R.string.quickpla
y_button);
        quickplaybutton.setEnabled(true);
        NetworkService.getInstance(this).update(n
ull, new QuickPlayEvent());
        }
/* @set Annotation.screenTransition=true;
@set Annotation.next=Annotation.LOBBYACTIVITY;
@set Annotation.previous =
Annotation.MAINMENUACTIVITY; */
Intent intent = new Intent(this,
LobbyActivity.class);
NetworkService.getInstance(this).update(null, new
JoinLobbyEvent());;
startActivity(intent);
}
```

**Figure 6. The expanded specification of `activityTransition` used in Figure 5.**

```
//@activity ("MainMenuActivity",null,"MainMenu")
public class LobbyActivity extends
FragmentActivity implements Observer,
ReceiveInviteDialogFragment.InviteDialogListener,
ErrorDialogFragment.ErrorDialogListener {…}
```

**Figure 7. Example use of the `activity` specification clause.**

```
/* @requires Annotation.screenTransition==true;
@requires Annotation.next ==
Annotation.LOBBYACTIVTY
@requires Annotation.previous ==
Annotation.MAINMENUACTIVITY;
*/
protected void onCreate(Bundle
savedInstanceState) {…}


/* @requires Annotation.screenTransition==true;
@requires Annotation.next ==
Annotation.LOBBYACTIVTY
@requires Annotation.previous ==
Annotation.MAINMENUACTIVITY;
@ensures this.hasWindowFocus();
*/
protected void onStart(){
//@set Annotation.screenTransition=false;
super.onStart();
}

/* @requires Annotation.screenTransition==false;
@requires this.hasWindowFocus();
@ensures Annotation.next ==
Annotation.MAINMENUACTIVITY
@ensures Annotation.previous ==
Annotation.LOBBYACTIVITY;
@ensures Annotation.screenTransition==true; */
public void onBackPressed(){
/*
@set Annotation.screenTransition=true;
@set Annotation.next =
Annotation.MAINMENUACTIVITY;
@set Annotation.previous =
Annotation.LOBBYACTIVITY;
*/
super.onBackPressed();
}
```

**Figure 8. The expanded specification of the `activity` clause used in Figure 7.**

### *5.3.1 Expanding activityTransition and activity*

Though an implementation of activityTransition and activity does not need to exactly match the expansions in Figures 3 and 5, it should not be significantly semantically different.

The `activityTransition` clause contains information about which Activity will be the next. The expanded specification of `activityTransition` should include the following:

- The system should not already be transitioning when initiating a transition. In Figure 6 this is given by the requires clause stating that the ghost variable `screenTransition` is false.
- The initiating `Activity` should currently have window focus.
- The completion of this method should ensure that the ghost variables used for transitions have been updated correctly. In Figure 6 this is given by the ensures clauses that `LobbyActivity` is next and `MainMenuActivity` is previous.
- After this method is finished the intent will be handled which means the system should currently be in a transition.
- Before the intent itself is called the ghost variables used for transitions should be updated.

The `activity` clause contains more information about its place in the Activity transition process. The expanded specification of `Activity` should contain the following:

- When calling both the `onCreate` and `onStart` methods the system should currently be in a transition.
- In both the `onCreate` and `onStart` method the ghost variable, indicating what the next Activity should be, should match the Activity being specified.
- In `onCreate` the ghost variable, indicating the previous Activity, should match one of the Activities given in `launchingActivities`.
- In `onStart` the ghost variable, indicating the previous Activity, should match one of the Activities given by `launchingActivities` or `backingActivities`.
- After `onStart` has finished the `Activity` should have window focus.
- The expanded specification for `onBackPressed` should be similar to `activityTransition`. The possible Activities `onBackPressed` can transition to is given by `backableActivities`.

Though not included in our description or in the examples, the expansion for these two clauses can include assignable clauses which describe the framing.

## 5.4 Window focus

At the start of this section we made an assumption that only Activities with window focus should be able to launch new Activities. While this assumption will generally hold true this does not always need to be the case. To allow this behaviour we propose to overload the `activityTransition` semantic clauses with an additional boolean to allow the window focus requirement to be dropped in cases where window focus is not required. With this modification the activity semantic clauses change their meaning only slightly allowing for the window focus requirement to be skipped.

# 6. CONCURENCY THROUGH THE HANDLER CLASS

In this section we discuss a set of concurrency methods in Android and how to specify their behaviour using one of the concurrency methods as the example. The methods we discuss are the post methods from the Android `Handler` Class. These methods achieve concurrency by adding their message, usually a runnable, to a message queue which will be processed when a certain condition has been met. The most common use of these methods is the use of `postDelayed` to ensure a splash screen transitions to the next Activity after a certain amount of time has passed. We use `postDelayed` as our leading example.

We propose two new JML specification clauses: `OnCondition` and `tag`. In this section we discuss the intuition behind these two clauses and how both static and runtime checkers could verify these clauses. A complete version of its syntax and semantics is given in Appendix A.

## 6.1 The case study

The first `Activity` in our case study is `SplashActivity`. `SplashActivity` is a simple `Activity` which will display an image and after a certain amount of time has passed will transition to `LoginActivity`. While Android has several ways to implement such behaviour, the generally suggested way is using `postDelayed` to create a runnable which will create the intent for the next Activity. This is also how it was implemented in the case study.

## 6.2 Specifying using basic JML

The `postDelayed(Runnable r, long delayMillis)` method from the Android `Handler` class allows `Runnables` to be added to a message queue. The added runnable is then executed after a specified amount of time has elapsed. The `postDelayed` method and its related methods are difficult to specify using standard JML. A failed attempt of specifying the behaviour of a method using postDelayed is given in Figure 9.

```
//@ public ghost long calledTime;

/**
* This method is called by Android via the
onStart method. If successful this method will
launch LoginActivity after SPLASH_DURATION has
passed */
/* @ensures \result&& uptimeMillis()>=
calledTime+SPLASH_DURATION ==>
Annotation.next==Annotation.LOGINACTIVITY;*/
public boolean splashScreen() {
        boolean successfullPost;
        Handler handler = new Handler();
succesfullPost =handler.postDelayed(new
Runnable() {

        /* @public normal_behaviour
        @requires !mIsBackButtonPressed;
        @activityTransition("LoginActivity");
        @also
        @public normal_behaviour
        @requires mIsBackButtonPressed;
        */
        public void run() {
        finish();
        if (!mIsBackButtonPressed) {
        Intent intent = new
        Intent(SplashActivity.this,LoginActivity.
        class);
        SplashActivity.this.startActivity(intent)
        ;}}}, SPLASH_DURATION);
        //@set calledTime = uptimeMillis();
        return succesfullPost;
}
```

**Figure 9. Example of specifying code that uses `postDelayed`.**

For tools which perform runtime checking the specification in Figure 9 is almost the correct specification, however, it still has two problems both relating to the timing of the Activity transition. The first problem is that `LoginActivity` will not immediately become the next Activity because some processing still needs to be done which takes a variable amount of time. The second problem is that the post condition of this method is a conditional statement. The problem with this conditional statement is that once true its condition will remain true, but when a new `Activity` is launched `Annotation.next` is no longer `LoginActivity`.

Tools which use static checking however will have far more serious problems when attempting to verify this specification of `postDelayed` or other attempts. Unlike runtime checking where `run()` is called and the runtime checker can verify the specification, a static checker will not be able to enter the `runnable` and use the specification of run. This is because static checkers do not execute the code and will thus not add the runnable to the message queue (though depending on if the API has been specified will know that it has been changed, but not how), which results in static checkers effectively ignoring the contents of the runnable. In other words, static checkers are not able to make a direct connection between the current code and code that is called indirectly.

## 6.3 Specifying using extended JML

To allow correct specification of code that calls `postDelayed` or similar methods we propose two new specification clauses: `tag` and `onCondition`. `tag` is an identifying marker to be used in combination with clauses such as `onCondition` to allow those clauses to reference methods (and their specification) that are called indirectly. Tagging a method works by adding `tag(String tagId)` to a method declaration in the same was as `pure` is added. An example of using tag is given in the declaration of the Runnable in Figure 10. This tagged method can then be referenced in specification clauses by using the same `String` as `tagId`. The `onCondition(boolean condition, String id)` clause semantically means that when the condition becomes true or slightly after, the method referenced by id should be executed and that the specification of the referenced method should then hold. An example of `onCondition` used to specify `postDelayed` is given in Figure 10.

```
//@ public ghost boolean succesfullPost;
//@ public ghost long calledTime;
/**
* This method is called by Android via the
onStart method. If successful this method will
launch LoginActivity after SPLASH_DURATION has
passed */
/* @ensures succesfullPost ==>
onCondition(uptimeMillis()>=
calledTime+SPLASH_DURATION,id("run"));
*/
public boolean splashScreen() {
        boolean successfullPost;
        Handler handler = new Handler();
succesfullPost =handler.postDelayed(new
Runnable() {
        /* @public normal_behaviour
        @requires !mIsBackButtonPressed;
        @activityTransition("LoginActivity");
        @also
        @public normal_behaviour
        @requires mIsBackButtonPressed;
        */
        public /*@tag("run")@*/ void run() {
        finish();
        if (!mIsBackButtonPressed) {
        Intent intent = new
        Intent(SplashActivity.this,LoginActivity.
        class);
        SplashActivity.this.startActivity(intent)
        ;}}}, SPLASH_DURATION);
        //@set calledTime = uptimeMillis();
        return succesfullPost;
}
```

**Figure 10. Example of the use of onCondition.**

## 6.4 Checking onCondition

A runtime checking tool could implement the semantics of such a clause by adding instrumentation to the code to allow the checker to monitor when the condition becomes true and if the referenced method is called shortly after. Instrumentation is the ability to monitor a specific part, in code this is done by adding a code instruction which outputs the required information. An example of the output of a pseudo-runtime checker checking the example given in Figure 10 is given in Figure 11.

```
splashActivity.onStart called
        requires . . . true
        splashActivity.spashScreen called
                requires . . . true
                set calledTime
                ensures . . .
                        onCondition tag run
        ensures . . . true
uptimeMillis()>= calledTime+SPLASH_DURATION . . .
true at 6957 ms
#FA526B.run called
        tag run . . . onCondition at 6957 ms . .
        . current 7031 ms . . . accept
        requires . . . true
        splashActivity.startActivity called
...
```

**Figure 11. Possible output of a runtime checker verifying `onCondition`.**

A static checking tool however should implement onCondition differently. A similar implementation is not possible because, generally the method referenced in concondition will not be called directly. Because of this reason when an onCondtion become true a static checker should call the referenced method, thereby checking its requires and ensures clauses. An example of the in- and output of a pseudo-static checker checking the example given in Figure 10 is given in Figure 12. Unlike the runtime checker there is some additional difficulty in this example because the condition involves the elapse of time which a static checker typically does not concern itself with. A possible solution to this problem in a limited fashion involves adding the option to elapse time to the list of possible user interactions a static checker typically has.

```
splashActivity.onStart called
        requires . . . true
        splashActivity.spashScreen called
                requires . . . true
                set calledTime
                ensures . . .
                        onCondition tag run
        ensures . . . true
User interaction required to progress
HardwareButton back
HardwareButton home
Elapse time
User interaction: Elapse time
uptimeMillis()>= calledTime+SPLASH_DURATION . . .
true at 6957 ms
#FA526B.run called
        requires . . . true
        splashActivity.startActivity called
...
```

**Figure 12. In and output of a static checker verifying `onCondition`.**

It should be noted that as described above, this does not check the full functionality of `onCondition` because it assumes that during runtime the referenced method will be executed, though this may not be the case (that is, it may happen that the referenced method is never executed due to unforeseen constraints of the message queue). Additionally, in the examples given above there is an additional difficulty because time is is not directly changed by methods.

# 7. PERMISSIONS IN ANDROID

In this section we discuss permissions in Android and how to verify them. We propose one new JML specification clause which can be used to verify that all required permissions have been requested but also that no unneeded permissions have been requested.

It is generally desired that an application only requests permissions it needs to function. Research shows that many applications overestimate their requirements for permissions[3]. While on the surface this does not appear very difficult, it is lengthy to retrieve permissions when specifying. To retrieve permissions programmatically the `packageManager` from a `Context` class such as a `Activity` needs to be retrieved, from the `packageManager` the `packageInfo` of the permissions needs to be retrieved from which then finally the `String Array` of the requested permissions can be retrieved. An example of this is given in Figure 13.

```
/*
@ghost String[] permissionArray;
@set permissionArray =
getPackageManager().getPackageInfo(getP
ackageName(),
PackageManager.GET_PERMISSIONS)).reques
tedPermissions;
@requires (\exists int index;
(index>=0&&index<permissionArray.length
;
permissionArray[index].equals("android.
permission.DUMMY_PERMISSION");); */
public void requiresDummyPermission{…}
```

**Figure 13. Programmatically retrieving and checking permissions.**

To help verifying permissions we propose to introduce a new semantic clause: `permission (String permissionName)`. This clause abstracts retrieving permissions away and can also be used to verify an Android application. Using this clause both verification of the required permissions and no excess permissions can be done, without need for lengthy specification. First the tool implementing this clause should check that all permissions used in `permission` clauses are requested in the `AndroidManifest`. Secondly it should check that no unneeded permissions are requested by checking that each permission requested in the `AndroidManifest` is used in at least one `permission` clause. Mathematically speaking the set of all `permissionNames` used in `permission` clauses should be equal to the set of requested permissions in the `AndroidManifest`.

The section above assumes that specification developers use the `permission` clause correctly, only using the clause when the method does indeed require the permission. It is possible to verify native Android permission by use of the API which specifies what permissions methods require. A similar verification for non-native permissions however faces the same issue as the permission clause itself.

# 8. CONCLUSION

In conclusion this paper provides a basis for an extension of JML for the specification and verification of Android applications. It does so by providing several specification clauses which assist in specification and if implemented in a tool could be used for verification. The `activityTransition` and `activity` clauses can be used to describe the transition behaviour of `Activities`. The `onCondition` clause and its supporting clause `tag` can be used for specification and verification of the asynchronous concurrency methods such as `postDelayed`. Finally, permissions can be verified using the `permission` clause.

## 8.1 Future Work

The work presented in this paper can be continued in several ways. All specification clauses have only been developed on paper and testing and actual implementation still needs to be done. Only a small part of the Android platform has been covered in this paper and much still remains to be done. The specification and verification of other components could be researched. The transition behaviour of Activities can be further researched and how it interacts with the Activity Lifecycle. As mentioned in the preliminaries specification of the API is another possible research option related to the verification and specification of Android applications.

# 9. ACKNOWLEDGMENTS

My thanks to Wojciech Mostowski for the guidance provided.

# 10. REFERENCES

[1] System Permissions | Android http://developer.android.com/guide/topics/security/permissions.html Retrieved 09-06-2014

[2] Drechsel A., Vlutters S., Huang S., Ding H. and Steen J. Uitbreiding OVPong. Report for design project. Fall 2013

[3] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android Permissions Demystied," in Proceedings of the 18th ACM Conference on Computer and Communications Security, 2011.

[4] "Gartner Says Sales of Mobile Devices Grew 5.6 Percent in Third Quarter of 2011; Smartphone Sales Increased 42 Percent". November 15, 2011. Retrieved 09-06-2014.

[5] Hoare, C.A.R . 1969. An axiomatic basis for computer programming in Communications of the ACM. volume 12, number 10, pages 576-580. ACM Press

[6] The Java Modeling Language http://www.eecs.ucf.edu/~leavens/JML/index.shtml Retrieved 09-06-2014

[7] Masoumeh Al. Haghighi Mobarhan. 2011. Formal Specification of Selected Android Core Applications and Library Functions. Master thesis, Chalmers University of Technology, Sweden.

[8] Meyer B. 1992. Applying "Design by Contract" in Computer. volume 25, number 10, pages 40-51. IEEE Computer Society Press

[9] Wojciech Mostowski. Fully Verified Java Card API Reference
Implementation. Proceedings of the Verify 2007 Workshop (associated with
CADE 2007), Bremen, Germany, July 2007, CEUR Workshop Proceedings.

[10] Mostowski W. and Poll E., C. 2010. Midlet Navigation Graphs in JML. In Post Proceedings of the 13th Brazilian Symposium on Formal Methods (SBMF 2010)

[11] OpenJML: http://opemnjml.org/ Retrieved 09-06-2014

[12] Schmitt, P.H., Ulbrich, M., Weiß, B.: Dynamic frames in Java dynamic logic. In:Beckert, B., March´e, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 138–152. Springer, Heidelberg (2011)

[13] Wal, Jelmer ter. 2013. Specification and verification of selected parts of the Java Collections Framework using JML* and KeY. Master thesis, University of Twente, Netherlands

[14] Shorthand – Wikipedia http://en.wikipedia.org/wiki/Shorthand. Retrieved 10-06-2014

# A. DESCRIPTION OF THE EXTENSION

In this paper several new specification clauses were proposed. This appendix summariseseach of these new clauses.

## A.1 ActivityTransition

The activityTransition clause is used as a shorthand for writing the specification associated with launching a new Activity.

When checking this clause the checker expands this clause to JML specification which specifies the transition.

Syntax

```
@activityTransition(String
targetActivityName)
@activityTransition(String
targetActivityName,boolean
windowFocusRequired)
```

`targetActivityName`: The name of the Activity which is being launched.

`windowFocusRequired`: If the associated method should only be called when the Activity containing the associated method has windowFocus.

## A.2 Activity

The activity clause is used as a short hand for writing the specification associated with the transition behaviour of an Activity as a whole.

This specification clause should only be used in the header of a class which has Activity as a base class.

When checking this clause the checker expands this clause to JML specification which specifies the transition behaviour of the specified Activity.

Syntax

```
@activity (<String launchingActivity
|   String[]   launchingActivities>,
<String backingActivity | String[]
backingActivites>,          <String
backableActivity     |     String[]
backableActivites>)
```

`launchingActivity | launchingActivites`: The names of the Activities which can launch this Activity.

`backingActivity | backingActivities`: The names of the Activities, which can transition to this Activity when the back button is pressed.

`backableActivity | backableActivities`: The names of the Activities, this Activity can transition to when the back button is pressed.

## A.3 OnCondition

The onCondition clause indicates that when the boolean condition is true, the method with the tag matching String id and its associated specification should be executed.

Syntax

```
@   onCondition(boolean   condition,
String id)
```

`condition`: The boolean which must be true when the method with the tag matching id is executed and when true the method will be executed.

`id`: String which matches the tag of a method.

## A.4 Tag

Declaration to a method to allow specification clauses to reference the method.

A tag must be unique. (depending on implementation this may be on class, package or global level)

A tag is declared in the same manner as pure.

Syntax

```
@ tag (String tagId)
```

`tagId`: The String which is used by specification clauses to reference the method.

## A.5 Permission

The permission clause has 2 functions.

A method annotated with permission require that the specified permission is granted to the application.

All permission set in the AndroidManifest should be used in a permission clause at least once.

As an additional layer of verification the method body can be checked if a API method requiring the permission is present if the permission is a permission native to Android.

Syntax

```
@permission (String permissionName)
```

`permissionName`: The name of the permission that the specified method requires.