Behavioral Specification of Android Apps

Neelam Soundarajan, Swaroop Joshi, Yan Xu
Computer Sc. & Eng., Ohio State University
{neelam, joshis, xuyan}@cse.ohio-state.edu

ABSTRACT

The importance of software running on mobile platforms has been increasing rapidly. Given the critical tasks that many of these "apps" are designed for, the importance of developing methods for precisely specifying their expected behaviors and testing against these specifications, is clear. In this paper, we report on our on-going work to address these problems for applications on the *Android* platform. We propose an approach to specifying the behavior of Android apps in which such information as the detailed *layout* on the display are abstracted away but the behavior that results when the app user presses specific buttons on the display or provides other input, are specified. The approach also allows us to specify the behavior that results when the system invokes various *lifecycle* methods. We also briefly consider how suitable test cases may be generated, based on these specifications, so that the app may be tested to identify potential problems.

1. INTRODUCTION

The number and importance of applications running on mobile platforms has exploded in just a few short years. From banking to e-commerce, from location-based targeted advertising to even enabling social revolutions, "there is [indeed] an app for that". Unfortunately, the quality of the software varies a lot from one app to another. An important reason for this, we believe, is the lack of techniques to precisely specify the expected behaviors of these applications and methods to test that the apps do behave in the expected ways. In this paper, we report on our on-going work to specifying the behavior of *Android* apps [1] in which such details as the *layout* on the display are abstracted away but the behavior that results when specific buttons (on the display) are pressed or the app user provides other input, as well as the behavior that results when the system invokes various *lifecycle* methods, are specified. We then consider how suitable test cases may be generated based on the specifications.

The *Java Modeling Language* (JML) [3] has been widely used to define suitable models of Java classes and to provide specifications, in terms of the model, of the behaviors of the class's methods. Although Android uses a version of Java, there are important differences that JML does not account for; hence JML cannot be directly used to specify Android apps. First, control in an Android app flows among various *listener* methods corresponding to specific inputs that the app user provides by interacting via the *widgets* on the display rather than method calls that appear in the code of a typical Java program. The listener methods that are executed may, in turn, display appropriate information through particular widgets on the display or modify the internal state or both. Thus, our approach has to enable us to relate the sequence of user inputs via the display widgets with the sequence of outputs produced and the changes in the internal state. Further, the action of a listener method executed in response to a particular input may also result in the current *activity* being moved to the *back-stack* and another activity becoming the current one that then receives the subsequent user inputs, and our approach has to account for this as well.

Second, *lifecycle* methods play a critical role in the behavior of Android apps. The Android system, rather than specific lines of code in the app, is responsible for invoking the correct lifecycle methods at the correct times. Indeed, some user actions such as rotating the device, that the user may not even think of as inputs, can result in the system invoking one or more of these methods. These methods can have a substantial impact on the behavior exhibited by the app and our approach has to account for them and the relation to the sequence of user actions, in specifying the app's behavior. Third, the association between particular listener methods and specific *widgets* on the display is itself defined by the code in `onCreate()`, one of the lifecycle methods. Moreover, some of the information defined in the *resource* files also affect the behavior of the app. Therefore, our approach to formalizing app behavior must account also for these factors.

In spite of these differences, some important ideas of JML turn out to be quite useful for our work. For example, allowing the *model* of a class to have multiple components that can be separately modified rather than, as is the case with techniques based on the idea of *data abstraction*, makes it possible to define appropriate models for apps. Similarly, the idea of *ghost* fields, as distinct from model fields, makes it possible to include the sequences of input and output events as ghost fields and to include, in the specifications, their relation to the way that the (model) state evolves. Hence in the next section, we briefly summarize some of these features of JML.

In the next section, we also summarize the main aspects of Android[1]. In general, an Android app is made up of four

[1]The name "Android" is generally used to describe both the underlying system as well as the language notation of the apps. We follow this practice; the context should make clear which is intended.

different types of components. Most apps, however, define only one type of component, that being *activity* (although these apps generally make use of other types of components that are defined as a standard part of Android). In the current version of our formalism, we only deal with such apps. Moreover, in our formalism, we currently ignore the notion of *fragments* which are used especially in apps designed to run on devices, such as *tablets*, with larger screens; hence in the summary in the next section, we do not consider the fragment mechanism. Further, although Android allows an activity to be defined by inheritance from another activity defined in the same app (or, possibly, a different app), most apps define their activities by inheriting from the standard Activity of Android and we only deal with such apps in our formalism[2]. We illustrate the main aspects of Android with *GeoQuiz*, a simple app borrowed from a standard textbook [7] on Android.

In Section 3, we present the details of our approach. We first consider how the model of an activity may be defined. Next we consider the form of specifications we use to express the behavior of the activity in terms of the model. As we will see, our approach allows us to deal with apps that have multiple activities with the effect of the various lifecycle methods being accounted for suitably. We illustrate the approach by applying it to the GeoQuiz app. In Section 4, we summarize our approach, consider the numerous additional aspects of Android that need to be accounted for. We also consider how an app may be tested against our specifications.

2. BACKGROUND

2.1 Java Modeling Language

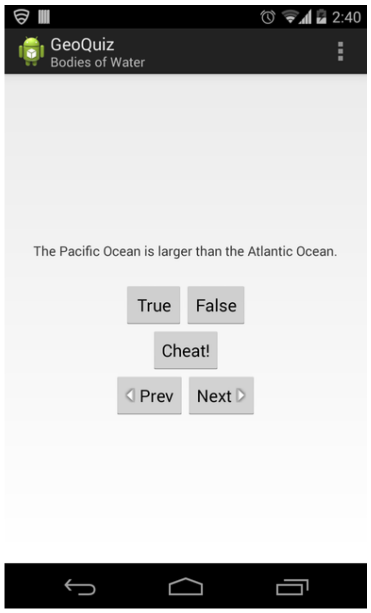
A key idea underlying JML is that the specifications should be written in a notation that is consistent with Java and that it should be possible to interleave JML specifications with Java code in a natural manner, enabling programmers to work comfortably with such specifications. This also makes it possible to extend Java tools to handle code that includes specifications. Thus, for example, *jmlc* is an extension of the standard compiler that, given such code, produces byte code that checks that various specified assertions, such as pre- and post-conditions as well as class invariants, are satisfied at various points during execution [3]. We plan to retain this idea in our work although the current version of our specification notation, described in Section 3, may need to be revised in order to help the building of useful tools.

JML is a model-based specification technique. Thus, in specifying a Java class, we define a *model* that is appropriate from the point of view of the class's *client*. A set of rep clauses specifies how the actual member fields of the class map to the model fields. The model of a class may consist of a number of more or less independent components that disjoint sets of members fields of the class map to. In addition to the model, JML also allows for *ghost* fields. The distinction with model fields is that ghost fields allow one to capture *additional* information beyond what is available in the member variables, i.e., in the state, of the class. This may include, e.g., the history of prior method calls.

Method specifications as well as class invariants in JML may be written using calls to *pure* methods, i.e., methods that do not change the class state. These pure methods provide convenient access to specific components of the model or useful information about them. In addition, JML allows us to update the ghost fields by introducing set statements in the code of the class. These are needed because these fields, as noted above, contain information that is not in the state variables of the class. Hence unlike regular model fields they will not have their values suitably updated by the actions of the original code in the class. In the next section, we will consider a possible alternative to this approach to updating ghost fields, at least the ones that we use in our model.

2.2 Android System (Simplified)

Android is a powerful system consisting of the underlying operating system, a large framework with a number of libraries useful for building apps, the *Dalvik* virtual machine that executes the compiled apps, etc. We will not consider those details, focusing instead on the structure of the apps. An app may consist of four different types of components, these being *activities*, *services*, *broadcast receivers*, and *content providers*. Most apps consist only of activities although, as we noted earlier, apps typically use other types of components that are provided by the Android system (or other standard apps). In this paper, we will focus on activities. In order to keep the discussion simple, we will use, as our motivating example, the *GeoQuiz* app, borrowed from [7]. This app consists of two activities, the main one called QuizActivity and a second one called CheatActivity. Fig. 1 shows the screen corresponding to QuizActivity, the main activity of this app. The user interface (UI) for this activity includes five *widgets*; the first is a *text-box* that poses a simple geography-related true/false question. The next two widgets, labeled True/False, are buttons that can be clicked by the user to answer the question, after which the activity will display an appropriate response in the form of a *toast* ("Correct!" or "Incorrect!") message[3]. If the user clicks the button labeled Next, respectively Prev, QuizActivity displays the next, re-



The Pacific Ocean is larger than the Atlantic Ocean.

Figure 1: GeoQuiz: QuizActivity

[2]Dealing with activities defined as derived classes in this manner would, as in the case of dealing with inheritance in Java, require us to deal with problems related to the introduction of new member variables in the derived class and its impact on *behavioral subtyping*. The approach of using *data-groups* [6] as in JML can be applied to address this problem.

[3]A *toast* is a message that is displayed briefly and then disappears.

spectively previous, question. The Android system is event-driven with much of the action taking place in the *listener* methods that the system executes when particular events, such as the user pressing the True button, take place. The mapping of the listener method corresponding to each possible event is typically specified in onCreate(), one of the main lifecycle methods. In more detail, such items as the appearance of the various widgets and the overall appearance of the UI, are specified in `layout.xml`, one of the *resource files* of the app; the onCreate() method attaches, using the *id*'s of the widgets defined in the resource files, the specific listener method for the event corresponding to each widget to the particular widget [4].

The button labeled Cheat! in Fig. 1, if clicked, (fires its listener method which) invokes the the second activity, the screen for which is shown in Fig. 2. This activity allows the user, by pressing the button shown, to take a peek at the answer before going back to the original activity, by pressing the standard Back button, and answering the question! But this activity, when it returns to the original activity, returns information about this so that it can adjust its response to the True /False buttons being clicked. When the user originally pressed the Cheat! button, the triggering of the second activity takes place because of a startActivity() command in the button's listener method. Or, more accurately, there is a startActivityWithResult() command that triggers CheatActivity since, as just noted, this second activity will return information to QuizActivity. When this command is executed, the Android system prepares to *pause* the current activity by executing its onSaveInstanceState() and its onPause() lifecycle method. When the activity resumes, the system executes its onResume() lifecycle method. The default definitons of these methods ensure that information about the views, etc., of the activity are saved and restored. When the Back button is pressed when the CheatActivity is running, its onPause(), onStop(), and onDestroy() lifecycle methods are executed before the system resumes QuizActivity by executing its onResume() method.

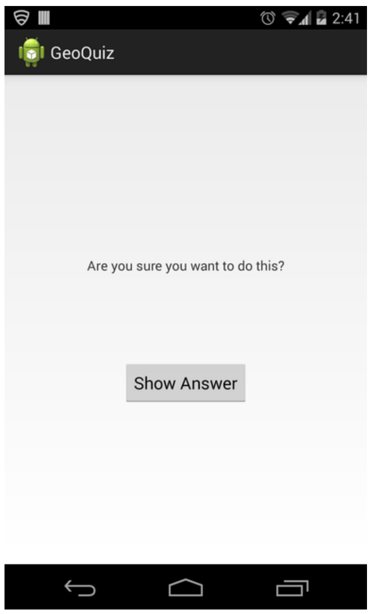


Figure 2: GeoQuiz: CheatActivity

Suppose the user peeks at the answer and then, before pressing the device's Back button to return to the original activity, *rotates* the device. In that case, Android *destroys* the activity and *re-creates* it. Moreover, it cannot simply resume QuizActivity by executing its onResume() method since the layout has changed (in general, this applies to any change in *configuration* of the device since the screen resources etc. have to be recomputed based on the new configuration). Instead, it has to first destroy the activity and then re-create it by executing its onCreate() followed by execution of its onResume() method. But then, as Philips and Hardy [7] note, unless the code they provide is appropriately corrected, the user will be able cheat with impunity! The goal of the specification formalism we develop in the next section is to capture such subtleties[5].

3. BEHAVIORAL SPECIFICATIONS

3.1 Models of activities

The model of an activity, in our approach, consists of three distinct components. First is the model of the activity's user interface. While it would be possible to work with the actual UI of the activity, this would contain far too much detail concerned with the appearance of each widget such as its color, size, etc. Moreover, nearly all apps use the standard set of widgets provided by Android and it would be pointless to repeat this information in the specification of each one. Therefore, in our approach we include a number of predefined *model types*, one corresponding to each type of widget that may appear in the UI. These include *MButton*, *MTextView*, and *MEditText*, corresponding, respectively, to widgets of type *Button*, *TextView* and *EditText*. An instance of MButton will include just one piece of information, its *label*, such as "`True`" or "`Cheat!`". An instance of MTextView will have a single piece of information associated with it, the string of text displayed in it–or rather in the corresponding Android TextView. An instance of MEditText will be similar, the difference between the two, reflecting the difference between widgets of type TextView and EditText, being that while the string in an instance of MTextView is modified by means of the *setText*() operation (acting on the corresponding TextView widget), the latter can also be modified by the user; or, more precisely, by the action of the corresponding listener method on the EditText widget.

The second component of the model of an activity, as in JML's model of a class, consists of a suitable set of model fields whose values can be used to represent the state, i.e., the concrete values of the variables of the particular activity. In simple examples, such as the two activities of the GeoQuiz app, this part of the model will consist of the same variables as the ones in the actual activity.

[4]Two points should be noted. First, Android does not *require* this. It is possible to define the listener methods associated with the events in other ways. But almost all apps follow this approach and it is considered a best practice; our formalism is based on the assumption that it, as well as other standard best practices, are followed. Second, some widgets may have multiple events associated with them; e.g., a button widget may be able to handle a regular *press* as well as a *long-click* with the resulting behavior being different for these two events. Our approach, as we will see in the next section is capable of handling this although, in much of our discussion, we will not explicitly consider it.

[5]The change in the app needed to deal with this bug requires us to modify the code of both activities to save in the *bundle*, by overriding their onSaveInstanceState() methods, information about whether the user has cheated so that this information is restored by the onCreate() method when the activity is recreated.

The third component of the model of an activity is used to represent important events, both those involving user input via the widgets on the UI and user actions such as device rotation; as well as events such as calls to lifecycle methods, triggered by the Android system. We split this part of the model into two pieces, the first one which we will denote $\nu()$ which gives us the *next event* that the activity has to respond to. This event may be either a user input event such as the user pressing the Cheat! button which will result in the triggering (by Android) of the corresponding listener method; or an event denoting, e.g., return from another activity such as return from CheatActivity to QuizActivity which will result in the triggering of the onActivityResult() method of QuizActivity and then its onResume().

The other piece of this component is a *sequence*, which we will denote σ, that represents all the *past events* related to the activity. In fact, given that an app may have more than one activity, we have multiple such sequences, one corresponding to each activity. If it is not clear from the context, we will use subscripts such as, e.g., σ_{c_a} or σ_{q_a}, to distinguish between the sequence associated with CheatActivity from that associated with QuizActivity. In a sense, each event "moves" from $\nu()$ to the σ (of the current activity); but an event $\nu()$ may result in Android invoking several life-cycle methods and each of these, and the original event from $\nu()$ will appear in σ. We also use σ_σ to denote the sequence of *all* past events involving *all* of the activities of the app, essentially an appropriate *merge* of the individual σ's. Being able to easily refer to the next event and the sequence of past events makes it convenient, as we will see, to specify the relation between the state of the activity and the events it has been involved in. Moreover, $\nu()$ allows us to abstract away all of the considerable work that the Android system has to perform in converting actual user actions, including complex ones such as multi-touch gestures, into appropriate input events and forwarding them to the appropriate listener methods. Other kinds of interactions with the user that the device might allow for, such as sounds or vibrations etc., can also be accounted for in a straightforward manner by $\nu()$ and σ. We only need to introduce additional model types, similar to MEditText, that abstract away all but the key information exchanged, in each such interaction type, between the user and the device and that is relevant to the app. $\nu()$ and σ are, in JML's terminology, *ghost fields* since they are not a model of information contained in the activity's variables.

Before concluding this section, it may be useful to consider the following question. Suppose $\nu()$ returns, as the next input event, (True, $\langle\rangle$), indicating that the True button was pressed, the second part indicating that there is no additional associated data. How can we be sure that QuizActivity is ready to process this event? For example, what if the activity is currently *paused* because it invoked CheatActivity and there was no event indicating return to QuizActivity? This cannot happen, of course, because the Android system will ensure that such events can take place only when QuizActivity is running. But how is this represented in our *formalism*? The answer is that we will capture such properties in a set of *Android axioms*, some of which we will see below[6].

[6]We should note that the set of these axioms is very much a work-in-progress and we welcome comments from interested researchers to help us fine-tune the axioms.

3.2 Specifying behavior

When specifying a Java class in JML, we specify its invariant and the pre- and post-conditions of each method as assertions over the model fields of the class[7]. The class invariant is required to be established by the constructor and maintained by each method. For an activity, its onCreate() lifecycle method plays a role similar to that of the constructor but it is possible that the activity was previously running and for some reason, perhaps because the device was rotated, was destroyed and is being *re-created* by the system, using the *bundle*, if any, that was previously saved by its onSaveInstanceState(). Thus, to allow for this, our *activity invariant* will be a relation involving σ and the rest of the model of the activity. And, as in the case of a Java class where each method is expected to preserve the class invariant, each listener method of the activity should preserve the activity invariant. Here too, the fact that the activity invariant can refer to the elements in σ helps with expressing properties suitable for inclusion in the invariant. This is because the conditions satisfied by the (model of the) member variables of the activity will depend on the particular events that the activity has gone through, including any values that the activity may have received in those events, as well as any values that the activity may have communicated to the outside world in some of the events[8].

There is another essential aspect of the behavior of the activity that has to be specified as part of its invariant. This has to do with which listener method is attached to which widget on the UI. This connection is established as part of the onCreate() method, almost always as follows. For each widget, get a reference to it via its *resource id* (as defined in the resource files) using the Android standard function, findViewById(); and assign the reference to a specific member variable of the activity. Next, use the setOnClickListner() method on that widget to specify the code to be executed when the particular widget is clicked. And if the widget can receive more than one kind of event, use setOnClickListner() repeatedly to attach multiple listeners to the widget. Within the rest of the code of the activity, the widget is accessed via the reference to it in the particular variable of the activity. Although, in principle, these assignments can be revised during the life of the activity, standard practice followed by nearly all apps is to retain the correspondence for the life of the activity; in this paper we will assume that is the case.

In the activity's invariant, we thus include, for each widget in the activity's UI, the following information: the model type of the widget, e.g., MButton in the case of a button widget or MEditText in the case of a text box that can accept text input from the user; the program variable that references the widget; and the specification of the listener method bound to the widget. As we noted earlier, calls to these methods don't appear explicitly in the app's code. Instead, Android invokes the methods when the corresponding events happen (via the UI). So how do we ensure that the pre-condition of such a method is satisfied when it is invoked and what do we do with its post-condition?

[7]More precisely, only the specifications of the *public* methods are to be available for use outside the class. But we ignore these issues in this paper since essentially the same approach to handle them as in JML is appropriate for our context.
[8]It is worth noting the parallel here to the approaches commonly used to deal with interacting processes in approaches such as CSP [5, 8].

Consider the Next button in Fig. 1. Suppose we are already at the *last* question in the app. One possiblity in this case, if the Next button is pressed at this point, would be to "cycle around" and go back to the first question. But suppose the app designer, instead, has decided to *disable* the button in this situation (by calling setEnabled(false) when the last question is reached). This relation between where we are in the question array and the value of the *enabled* bit in the state of the button can be easily represented by including a suitable clause in the activity's invariant. And, in the pre-condition of the listener method bound to this button, we require this bit to be *true*.

The question at the end of the last paragraph may then be stated, in this situation, as, how do we ensure that the *enabled* bit of the button (or rather of its state) will be true when its listener method is invoked? What happens in the actual system is that once setEnabled(false) is invoked on a given button, Android *greys out* that button's display on the UI and the user's attempts to click on it are ignored; and this continues until setEnabled(true) is invoked on the button. But how do we represent this in the formalism? Since this is behavior that the Android system, rather than the individual app, is responsible for, we represent it via the following Android axiom:

$$[(\nu().wid = b) \wedge (b.\mathrm{Type} = \mathrm{MButton})] \Rightarrow (b.enabled = true) \tag{A0}$$

(A0) states that if the type of the widget (*wid*) in the next input event is MButton then its *enabled* bit must be *true*. This axiom, in conjunction with the activity's invariant will allow us, in the case of the scenario described above, to show that when the Next button is invoked, we are not currently at the last question in the question array and hence the index can indeed be advanced. Given this axiom, the precondition for the listener method of most widgets would be simply *true* since, if a particular widget is not greyed out, the user may indeed click on it and the listener method must be able to handle the event[9]. As far as the post-condition is concerned, we simply need to ensure that when the method finishes, the activity's invariant is satisfied.

In addition to the listener methods, the activity may define a number of *lifecycle* methods which the Android system invokes at appropriate points. Android provides default definitions for the lifecycle methods. If an activity provides a definition for a lifecycle method, it will, as a rule, consist of a call to the default definition provided by Android, followed by additional code to achieve needed app/activity-specific additional effects. Although Android includes a large number of lifecycle methods, the ones that are important for most apps are, onCreate(), onPause(), and onResume(), and in this paper, we will not consider others. Android invokes onCreate() when the activity is created or re-created; it invokes onPause() when the activity is going to the background and another activity is getting control of the display (and will interact with the user); it invokes onResume() when the activity is coming to the foreground (either for the first time or after having been paused) and will interact with the user.

[9]If the pre-condition of a listener method is stronger than the activity's invariant (plus requiring $\nu()$ being of the appropriate type, i.e., the event being matched to this widget, and the *enabled* bit of the widget being *true*), that may indicate a potentially serious problem since it means that the app may behave in unspecified ways if the user were to try to use the particular widget when the additional conditions that are part of the pre-condition are not satisfied.

There are a couple of other methods that are somewhat related to the lifecycle methods that are important for many apps. An activity may define onSaveInstanceState() to save, on the standard *bundle*, the values of important variables (such as the current value of the index in the question array of QuizActivity). Android will invoke this method if it destroys an activity, perhaps because the device was rotated. The activity can then use the saved bundle in its onCreate(), when Android invokes that method when the activity is re-created, to restore these variables. An activity may use startActivityForResult() to call another activity (expected to return a result); e.g., QuizActivity uses this method to invoke CheatActivity when the user presses the "Cheat!" button. The invoking activity should also define onActivityResult() which Android will invoke (immediately before invoking onResume()) when it resumes the current activity after the invoked activity finishes.

Our formalism has to account for the behavior of these methods and we will consider some of the issues below.

3.3 Behavior of GeoQuiz

Many apps, including GeoQuiz, make use of *resource files* to specify such things as string constants, rather than polluting the code of the individual activities with this information since even for small apps, the amount of this type of information can be substantial. Thus, in GeoQuiz, `string.xml` is defined to contain, as strings, the various questions that might be posed to the user as well as other string constants such as the message that might be displayed if the user provides the correct answer; a unique name is also specified for each string. Android then generates a class file (R) that contains these string constants and their names. Within the individual activities, as part of the initialization code, a set of variables are, typically, assigned the values from the R class. This substantially reduces the size of the initialization code of these activities. Clearly, it would make sense to apply this idea to specifications. We do so by assuming that A(R) is a set of assertions that specifies these constants and their names and making A(R) available when reasoning about the app. For example, corresponding to the line,

```
⟨string name="q_1"⟩
  Source of Nile River is in Egypt⟨/string⟩
```

A(R) will contain the assertion (q_1 = `"Source of Nile..."`) This will allow us (when checking/verifying the code against our specification) to conclude that an assignment such as "`X=R.string.q_1`" will result in X containing the specified string. Two points should be noted. First, A(R) can be thought of as an Android axiom since it represents a specific functionality that Android implements; but unlike the axiom (A0) we saw earlier related to ensuring that a disabled button does not receive any input, the set of assertions in A(R) varies with the contents of the resource file(s) and hence the app. Second, not just strings but other types of constants along with unique idenifying names can and are usually specified in resource files in most apps; and the class R includes all of them. Even the `layout.xml` file which specifies the layout of the UI and the activity's widgets is a resource file and the information is represented in R. The onCreate() operation associates suitable listener methods with each widget (identified by its unique name specified in the layout file) and we have to specify the behavior of the listener method associated with each widget as part of our specification of the activity and we turn to that next.

Consider the behavior of the button labeled Next in QuizActivity. Recall that, according to the activity's invariant, this button would be *disabled* if the current value of the question array index, mCI, is already at the last element in the question array, mQA; and that (A0) ensures that in this case, $\nu()$ will not be a click on this button, hence we do not have to worry about this possibility in specifying its listener method. The behavior of the listener method associated with this button may be summarized as follows: increment mCI by 1; replace the display in the *TextView* box on the UI with the question which is the value of mQA[mCI][10]; and grey out the Next button if the new value of mCI is at the last element of the array.

In terms of our model, a listener method, in general, has three sets of effects. First, it moves the input event that resulted in its invocation to σ, the history sequence associated with the current activity; second, it updates the (model) variables of the activity appropriately; third, it may add one or more additional elements to σ corresponding to other resulting events. The first effect does not have to be included in the specifications of individual listener methods since it represents the behavior of Android rather than of the particular listener method; hence we represent it in our formalism by an Android axiom (which we omit). We will use δ to denote the elements added to σ, i.e., the change in σ, as a result of the execution of this method. Thus the post-condition of the listener method associated with clicking the Next button may be written as follows:

$$\begin{aligned}
&\mathit{ensures}_{\mathsf{clickNext}} \equiv && (1)\\
&\qquad [\ \ (\mathsf{mCI} = \mathsf{mCI@pre} + 1)\\
&\qquad \wedge (1 \leq |\delta| \leq 2)\\
&\qquad \wedge (\delta[1] = (\mathsf{mQTextView}, \mathsf{update}, \mathsf{mCQ[mCI]}))\\
&\qquad \wedge ((\mathsf{mCI} < |\mathsf{mCQ}|) \Rightarrow (|\delta| = 1))\\
&\qquad \wedge ((\mathsf{mCI} = |\mathsf{mCQ}|) \Rightarrow ((|\delta| = 2)\\
&\qquad\qquad \wedge\, \delta[2] = (\mathsf{mNButton}, \mathsf{setEnabled(false)})))]
\end{aligned}$$

The first clause specifies the increase in the value of mCI compared to its pre-value. The second clause states the length of δ lies between 1 and 2, i.e., one or two elements (in addition to the standard input event) will be added to σ_{q_a}. The third states that the first element added to σ_{q_a} will be an update of the question displayed in the TextView referenced by the variable mQTextView. The last clause states that if mCI is less than the length of the question array, only one element is added to σ, else two elements are added with the second element being one that disables, i.e., *greys-out*, the button referenced by the variable mNButton.

But how exactly are the elements of δ, i.e., the new elements of σ, created? In JML, given that σ is a ghost variable, i.e., there are no actual variables of the QuizActivity class that map to σ but instead it represents additional information that is not saved in any of QuizActivity's variables, we are required to insert suitable set statements in the code to appropriately create these new elements. In our approach, we adopt the following alternative method: in our reasoning system, the axiom for the statement mQTextView.setText(. . .) that appears in the code of the activity, has the effect of creating this δ element. We can adopt this approach because we don't, unlike JML, allow for arbitrary ghost variables. The only ghosts in our approach are ν and σ (and δ which is simply a notational convenience denoting the new elements of σ). If we had allowed arbitrary ghost variables, then our formal system could not adopt a single approach such as the above to update all of them; and we would need to allow set statements[11].

Consider next the behavior of the button labeled True. When this button is clicked, as we saw in Section 2, a *toast* message appears. One of four different messages may be displayed, these being "Correct!" and "Incorrect!" depending on whether the answer matches what is in the array of QuizActivity for the question at the mCI index, provided the value of mIsCheater whose value denotes whether the user peeked at the answer to the question before pressing the button is false; and if mIsCheater's value is true, the corresponding messages are the ones in `strings.xml` associated with the appropriate id's.

$$\begin{aligned}
&\mathit{ensures}_{\mathsf{clickTrue}} \equiv && (2)\\
&\qquad [\ \ (|\delta| = 1)\\
&\qquad \wedge (\exists k : (((\mathsf{mCA[mCI]} = \mathsf{true}) && (2.1)\\
&\qquad\qquad \wedge (\mathsf{mIsCheater} = \mathsf{false})) \Rightarrow (k = 11))\\
&\qquad\quad \wedge \ldots\\
&\qquad\quad \wedge (\delta[1] = (\mathsf{Toast}, q_k)))]
\end{aligned}$$

The first clause asserts that one element will be added to σ. The line labeled (2.1) and the next one introduce a variable to make it easy to refer to the various messages that may be displayed. The two lines shown state that if the correct answer to the question is indeed *true* and the user has not cheated, the value of this variable is 11 since (we are assuming that) the id of the corresponding string in `strings.xml` is q_{11} so that, using the assertions in A(R), we can conclude that the correct toast will be displayed.

Let us now briefly consider the Cheat! button or, rather, what happens when the CheatActivity returns to QuizActivity. If the device is currently running CheatActivity, the user may indicate to the system that she wants to return to the previous activity by pressing the Back button (which is an Android facilty and is not app-specific). The code in CheatActivity ensures that information about whether the user peeked at the answer is saved (in the form of an *intent*) in the result to be returned to QuizActivity. When the user presses the Back button, the current activity is destroyed, and the saved result becomes available to the invoking activity which, in this case, is QuizActivity. Android executes the onActivityResult() of QuizActivity then its onResume(). In other words, if the value of $\nu().\mathit{wid}$ is $(\mathsf{Back}, \langle\rangle)$, the resulting δ will consist of two elements, onActivityResult(), followed by onResume(). What about onDestroy() on CheatActivity? That element will be added to the history of CheatActivity, σ_{c_a}, not to σ_{q_a}. All of this has to be captured in an Android axiom corresponding to the click of Back button since it is not app-specific.

Given specifications such as (1) and (2) which capture the behavior of the various widgets for each activity in the app and given the Android axioms which formalise the behavior the system provides, we can, given any sequence of input events, arrive at the sequence of events that the Android system can be expected to go through and the results that the user can expect to see on the display. We can then, e.g., *test* the app to see whether it meets our expectations.

[10]The actual question at this array element will, as we saw above, be the appropriate one defined in `strings.xml`.

[11]We should note that, in the specification (1), we have assumed that the (model) variables that are not explicitly referred to remain unchanged.

4. DISCUSSION

The goal of our on-going work is to develop a suitable formalism to enable us to precisely express the behaviors of Android apps. As we noted earlier, although Android apps are written in a version of Java, Android not only provides a number of facilities such as widgets that implement important behaviors and that are a key part of every app, the Android system implements specific behaviors when certain key events take place and neither of these have counterparts in Java. Thus an approach such as JML, as it stands, is not adequate for our purposes. Hence, in our work, we are developing suitable extensions to JML. Two novel aspects of our work have been, first, the use of ν and σ to capture, respectively the events that an activity may receive as input and the sequence of events it participates in, including events that the Android system injects into the execution at specific times. And, second, the introduction of Android axioms that represent the behavior that Android provides behind the scenes. These features allow us to abstract away much of the concrete detail that is not essential to thinking about the abstract behavior provided by apps.

We conclude with a point related to testing. *JUnit* is a widely used testing framework that is well suited for specification-based testing. It is especially suited for testing against JML specs since these specs are executable [4]. A key issue, though, is coming up with a relatively small set of test cases that *adequately* test the possible behaviors of the methods of the class, given various possible starting states, etc. *Korat* [2] takes a JML specification of a class and a set of parameters that, in some respect, limits the size of each test case, and generates such a set. It achieves this by ensuring that the starting state in each test case satisfies the class invariant and the pre-condition of the method in question; and by generating only non-isomorphic test cases, isomorphism being defined based on the structure of instances of the class. An important item of future work for us is to develop an approach, possibly similar to that of Korat, that allows us to test an app against its specification in our approach. An important question here will be how to define isomorphism between given sequences of input events.

5. REFERENCES

[1] Android. Android, the world's most popular mobile platform. http://developer.android.com/about/index.html, 2012.

[2] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *Proc. of ISSTA*, pages 123–133. ACM, 2002.

[3] P. Chalin, J. Kiniry, G. Leavens, and E. Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *Formal Methods for Components and Objects (FMCO)*, pages 342–363. Springer (LNCS), 2006.

[4] Y. Cheon and G. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *Proc. of ECOOP 2002*, pages 231–255. Springer-Verlag LNCS, 2002.

[5] C. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[6] K. Leino. Data groups: Specifying the modification of extended state. In *Proc. OOPSLA*, pages 144–153. ACM, 1998.

[7] B. Philips and B. Hardy. *Android programming.* Big Nerd Ranch, 2013.

[8] Q. Xu, W. de Roever, and J. He. Rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997.