Understanding, Specifying, and Testing Mobile Software

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

ABSTRACT

In the last few years, there has been an enormous growth in both the number of apps running on mobile platforms as well as in the *variety* of tasks that they are designed to perform. At the same time, the size and complexity of the code of even the simplest app can be daunting to someone trying to understand its behavior or is attempting to modify the behavior. In related on-going work, we have proposed an approach to the formal specifications of the behavior of Android apps. In this paper, we consider how, based on our formal specifications, we can arrive at precise documentation of an Android app; and an approach to testing the app and presenting the results in a manner that ties it to the documentation. We believe this will be of value to the original developer during development of the app as well as to others who might be interested in getting a good understanding of how it functions, perhaps with the goal of modifying it.

1. INTRODUCTION

The number and variety 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, apps have started dominating more and more domains. Indeed, all indications are that mobile software may well, in the near future, supplant both laptop and desktop software for nearly every type of application. At the same time, the size and complexity of the code of even the simplest app can be challenging to someone trying to comprehend its behavior, to relate specific aspects of the behavior with specific portions of the code, and possibly identify bugs in that code that may be responsible for the app exhibiting unexpected behavior or to modify its functionality in particular ways. A key part of the problem is that the code responsible for the *core functional* behavior of the app is often intertwined with code that is responsible for its appearance and other effects on the display. Thus the goal of our work is to develop ways that can be used to provide precise documentation of an Android app that focuses on its functional behavior; and an approach to testing the app and presenting the results of the tests in a manner that ties it to the documentation. This should be of value to the original developer during development of the app as well as to others who might be interested in getting a good understanding of how it functions, perhaps with the goal of modifying its core functionality.

In other on-going work, we are developing an approach to formally specifying the behavior of *Android* apps [2]. In that work, a preliminary version [10] of which has been submitted to the *ICSE '14 FormaliSE* workshop, such details as the *layout* on the display are abstracted away, the focus being on the formal specification of the essential functional behavior that results in response to specific user input. The main goal of the work reported in the current paper is to automatically generate, from these formal specifications, clear documentation of apps that is accessible to practitioners. Our formal specifications can also be used as a basis for creating suitable test cases for the app. A second goal of the current work is to generate understandable descriptions of the results of these test cases. As we will see, we use a widely used open-source tool for both tasks.

Before continuing, it may be useful to note that although Android uses a version of Java, there are important differences. First, control in an Android app flows among various *listener* methods corresponding to 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, in documenting the behavior of an app, we have to relate the sequence of user inputs received via the display widgets with the sequence of outputs that the app is expected to produce as well as the accompanying changes in its internal state. One especially important type of behavior is when the action of a listener method is expected to result in the current *activity* being moved to the *back-stack* and another activity becoming the current one that then receives and responds to subsequent user inputs. This has to be properly documented since the way the app responds to subsequent user inputs may completely change; and developers who wish to relate the exhibited behavior with specific parts of the app's code obviously need to know which activity is currently responding to user inputs.

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 of the activity 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 several of these methods. The execution of these methods can have a substantial impact on the subsequent behavior exhibited by the app as a whole. Hence, appropriate descriptions of these aspects of the app's behavior must be an integral part of its documentation. A key consideration in all of this, as noted above, is to ensure that the focus is on the core, functional behavior of the app,

rather than on such details as its appearance on the screen.

Similar comments may be made with respect to test cases and the results of running the tests. In other words, for each test case, we must generate a suitable description of the sequence of input actions and the resulting sequence of outputs as well as suitable information about the internal states of the various activities that might become current at various points along the way so that the developer can see whether the results agree with the expected behaviors.

These considerations mean that some of the existing approaches to the tasks of creating documentation of Java methods and classes and for creating readable descriptions of test cases are not suitable for our purposes. At the same time, as we will see below, ideas from some of these approaches can be suitably adapted and extended to allow us to meet our goals. In the next section, we briefly summarize some of these related approaches. We also summarize some of the important aspects of the Android[1]. In general, an Android app is made up of four different types of components. Most apps, however, define only one type of component, that being *activity* and we consider only such apps in this paper. In Section 2, we also briefly summarize our approach to formally specifying Android apps.

In Section 3, we consider our approach to creating precise and readable documentation of the activities of an app from their formal specifications. The approach exploits the *shadowing* facilities provided by *Robolectric* [7], an open-source tool designed for testing Android apps while running them on a standard JVM, rather than on an Android device or emulator. Robolectric enables this by defining *shadow* classes for the standard classes provided by the Android framework, with a custom class loader substituting instances of the shadow classes in place of instances of Android classes. The tool also allows its user to create custom versions of the shadow classes and it is this facility that will enable us to to generate documentation of activities from their specifications. It is the same facility that we exploit for producing readable summaries of test case results which we also describe in Section 3. In Sections 2 and 3, we use *GeoQuiz*, a simple app borrowed from a standard textbook [9], as an illustrative example. In Section 4, we briefly summarize our key ideas, our plans for implementation, and answer the questions posed in the ERA Track call for papers.

2. BACKGROUND AND RELATED WORK

Android System (Simplified):. We start with a brief summary of the Android system. It consists of the underlying operating system, a large framework of classes useful for building apps, and the *Dalvik* virtual machine that executes the compiled apps. An app may consist of four types of components, *activities*, *services*, *broadcast receivers*, and *content providers*. Most apps consist only of activities and, in this paper, we consider only such apps. The *GeoQuiz* app [9] consists of two activities, the main one called QuizActivity and a second one called CheatActivity. Fig. 1 shows the screen corresponding to QuizActivity, the app's main activity. The UI for this activity includes five *widgets*; the first, a *text-box*, poses a geography-related true/false question; the next two, labeled True/False, are buttons that can be clicked to answer the question, after which the activity will display an appropriate response in the form of a *toast* ("Correct!" or "Incorrect!") message. If the user clicks the button labeled Next, respectively Prev, QuizActivity displays the next, respectively previous, question. As noted earlier, the behavior in response to a user input is provided by the corresponding *listener* method. The mapping of the listener method corresponding to each event is (typically) specified in onCreate(), one of the main lifecycle methods. The code in this method attaches, using the *id*'s of the widgets defined in the *resource files*, the listener method for the event corresponding to each widget[2].

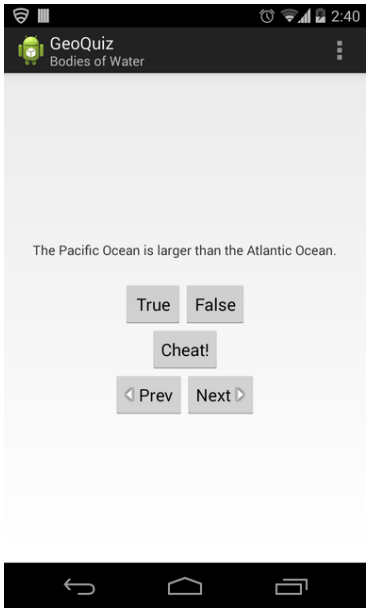

Figure 1: GeoQuiz: QuizActivity

The button labeled Cheat! in Fig. 1, if clicked, (fires its listener method which) invokes CheatActivity (whose screen we omit). CheatActivity allows the user, by pressing its ShowAnswer button, to take a peek at the answer before going back to the original activity, by pressing the standard Back button, and then answering the question! But CheatActivity, when it returns to QuizActivity, supplies information about this so that it can adjust the response to True/False buttons being clicked. When the user originally pressed Cheat!, the triggering of the second activity takes place because of a startActivityForResult() command in the listener method that triggers CheatActivity. When this is executed, Android *pauses* the current activity by executing its onSaveInstanceState() and then its onPause() method. When the activity resumes, the system executes its onResume(). All must be included in the documentation of the app; and violations of particular aspects of the expected behavior during execution of any test cases identified.

A number of authors have proposed techniques for generating documentation for methods as well as classes. Dragon *et al.* [4] propose techniques for automatically classifying individual methods as *accessor*, *creational*, etc., by static analysis of the code, relying on standard language structures and idioms. In [5], they extend the approach to classes, automatically categorizing given classes as *factory*, *controller*, etc., based on the frequency of occurrence of various method stereotypes obtained by their classification of the methods.

Moreno *et al.* [8] build on this work by adding human readable summaries of the class. These summaries are in-

[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.

[2]Android does not *require* this. It is possible to associate listener methods to events in other ways. But most apps follow this approach and it is considered a best practice. Second, some widgets may have multiple events associated such as *click* and *long-click*; these are two different events and typically have different associated listeners.

tended to allow the reader to get a broad understanding of the content and responsibilities of the class. The summary is generated by a tool and provides information about the stereotype of the class and its interfaces and superclass; a description of the class's structure based on its stereotype; a description of the behavior of its most relevant methods relevance being decided by its stereotype; and a list of inner classes (if any). In our work, we plan to adapt the approach of Moreno *et al.* to generate human readable descriptions from the *formal specifications* of an app's activities.

Kamimura and Murphy [6] present an approach to generating readable summaries of JUnit test cases for Java methods. Their tool looks for `assert*` calls to identify the assertion(s) the test case is intended to check. The tool identifies what is unique about a test case based on how many times it invokes various methods and with which arguments. Based on this analysis, the summaries generated identify the key methods that the test cases calls and the assertions it checks.

van der Merwe *et al.* [11] describe an extension of Java PathFinder, a model checking tool, to help detect problems such as deadlocks in Android apps. One similarity with our work is that their tool runs on a standard JVM, rather than the Android Dalvik machine. But their focus is on detecting *common* problems such as runtime exceptions whereas our focus is on the functional behavior that a given app is intended to exhibit. In some ways, their work is similar to that of using *Monkey* [1], a tool that (runs on Dalvik and) generates random user events to see if the app crashes.

We conclude with a brief summary of our on-going work [10] on formal specification of apps. We propose an extension of JML [3] which accounts for the main differences between Java and Android. First, corresponding to each type of widget that an activity might include on its UI, we define a model type that includes just the information related to its functional behavior; e.g., our model type *MTextView* has just one field, the string of text displayed in the corresponding widget; *MButton* has the label that appears in the corresponding *Button* as well as a boolean field that indicates whether the button is currently *enabled*, i.e., can be clicked; etc. Second, we introduce a sequence, σ_{ac}, that represents the history of all past events related to the activity *ac*. This includes both such events as events involving the UI as well as events generated by the Android system such as invoking onPause(). In addition, we use a function, $\nu()$, that specifies the *next* incoming event that the activity will have to deal with. Third, we define a set of *axioms* intended to represent the actions of the Android *system*; e.g., one of these axioms states that if the value returned by $\nu()$ is the click of a particular *mButton*, its enabled field must be *true*. Fourth, the activity invariant is written in terms of σ_{ac} and its model; the use of σ_{ac} makes it easy to capture important behavior that is related to the activity's past events. The specifications of the various methods are also written in terms of σ_{ac} and the model; in the post-conditions, we also use the notation δ to refer to the difference between σ_{ac} when the method started and when it ended, i.e., the sequence of events during the method execution.

3. SHADOWS TO DOCUMENTATION

Throughout this section, we use the *GeoQuiz* example to illustrate the discussion. We first consider the behavior of the listener method associated with the Next button of QuizActivity and its formal specification. Then we consider a potential bug in the method and test cases that might reveal the bug. Finally, we will turn to how we can exploit the *shadowing* facilities of *Robolectric* [7] for executing the test cases –without our having to actually write them in the form of JUnit test cases– and for generating documentation.

In our specification of the listener method of a widget, the pre-condition must be the assertion *true*, i.e., is always satisfied. The reason is that the user might click the widget at any time; and if we allowed a pre-condition that was not satisfied at the time of the click, the listener method may behave in an arbitrary manner (while being consistent with its spec). We avoid this by having *true* as the pre-condition.

The post-condition, or the *ensures* clause, of the listener for the Next button is as follows:

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

The first clause specifies the increase in the value of mCI, the (model) variable that holds the value of the index into the *question array*, compared to its pre-value. The second clause states that the length of δ lies between 1 and 2, i.e., one or two elements will be added to σ_{q_a}, the sequence that records the events of this activity. The third states that the first element added to σ_{q_a} will be an update of the question displayed in the *TextView* (widget corresponding to the *MTextView* model object) mQTextView with the newly displayed question being the one in the corresponding location of mCQ, the question array. The last clause states that if mCI is less than the length of mCQ, 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 Next button (mNButton being the corresponding model object).

In addition to specifications of various listener methods (and life-cycle methods), the specification of the activity will also include an invariant. One clause of the invariant for QuizActivity will specify the relation between the value of the array index, mCI, and the sequence of Next and Prev click events recorded in σ_{q_a}. Another will state that if this value of mCI is equal to the length of the array, then there will be a (mNButton, setEnabled(false)) event on σ_{q_a}; and the event that enables the button will appear following the next Prev event in σ_{q_a}. This, in conjunction with the Android axiom we briefly discussed at the end of Section 2 will ensure that the Next button cannot be clicked if we are already at the last element of the array. Thus each listener (and life-cycle) method is required to ensure that the activity invariant is satified when it finishes; and, in turn, we may assume that the invariant is satified at the start of the method.

What if there was a *bug* in the code of this listener method so that, in some cases, it fails to disable the button although the value of mCI is equal to the length of the array? In this case the app may fail since the user will be able to click the Next button, the Android system will invoke the listener method with the invariant not being satisfied at this point. Thus the question is, what test cases will help us identify such bugs and how do we arrive at them? The test case itself is fairly clear: generate a sequence of k events each of which is a click of the Next button, k being the length of the question array and assert that the (mNButton, setEnabled(false)) event appears on σ_{q_a} following the k^{th} click. More generally, *assert*, as part of the test case, that following the execution of each listener method, its *ensures*

clause as well as the activity's invariants are satisfied.

But, for this to work, we must ensure that the various variables referred to in the post-conditions and the invariant, including the sequence σ_{q_a} as well as model objects such as mNButton exist, and their values are appropriately updated at the correct moments. For example, (mNButton, setEnabled(false)) has to be appended to σ_{q_a} and the *enabled* bit of mNButton set to *false* when a method of QuizActivity invokes setEnabled(false) on the Next button. How do we ensure this when variables such as σ_{q_a} and mNButton are part of our *model*, not part of the actual Android app?

We achieve this by exploiting the *shadowing* capabilities of *Robolectric* [7]. As mentioned earlier, this tool allows tests for Android apps to be run on the JVM rather than on the Dalvik machine. It does by this defining *shadow* classes corresponding to the various classes provided by the Android framework; thus ShadowButton is the shadow for the Button class. A custom class loader which is part of the Robolectric *test runner* intercepts any call to create an instance of a class such as Button, replacing it with an instance of ShadowButton. Unlike an instance of the Button class which corresponds to a button on the screen of an Android device and has such properties as its color, coordinates, etc., with the Dalvik runtime system ensuring that the button displayed indeed has the right properties, an instance of ShadowButton is a simple Java object with few attributes.

Robolectric also allows us to *redefine* ShadowButton and other shadow classes as we choose (as long as certain inheritance relations are preserved). Given this, we redefine the ShadowActivity class to include the sequence σ of events that the activity is involved in; and redefine ShadowButton, ShadowTextBox, etc., so that methods such as setEnabled(false) when invoked on a Button, which the Robolectric *test runner* would have converted into an invocation on the corresponding ShadowButton, to not only set the *enabled* bit of the ShadowButton class to *false* –which the Robolectric definition already does– but also add the corresponding event to the *ShadowActivity*'s σ. Essentially, we define the various shadow classes to act as the corresponding *model* classes; and to include, in the definitions of the methods corresponding to the various events, not only the updates of the model objects but also add, to the sequence σ, elements recording the occurrence of the events – including internally generated events such as invocation of life-cycle methods.

While that is adequate for running test-cases such as the one we considered for identifying the bug in clickNext(), we go further. First, rather than writing the test-cases by hand, we write the code corresponding to the $\nu()$ function which, as we saw earlier, is intended, when called, to give us the next event. The sequence of events that this function returns when called repeatedly, in effect, gives us all the information needed for a particular test case; all we need then is to define the Robolectric "test-case" to instead be a test-case *driver* that repeatedly invokes this function and, based on the result returned, invokes the corresponding listener method.

Second, we include a file containing the specification of the app among the resource files and modify the *ShadowApplication* class so that it reads in this file. This file contains specifications such as (1) although we are still experimenting with the precise notation to be used. More importantly, following JML, rather than depending on a high-powered solver that can check whether or not a complex assertion is satisfied by the current values of the various (model) variables, we are identifying a small set of functions (such as the length of σ) and a restricted set of assertions based on these functions, which can be implemented easily. When the test-case driver invokes the various listener methods (as well as life-cycle methods), it is these assertions that will be checked following each such invocation. Further, borrowing from the work of Moreno *et al.* [8], we are developing simple templates that convert such assertions into readable summaries. For example, a complex assertion involving the length of σ and the values in particular elements of σ will be converted into a summary such as "appropriate condition involving the number of elements of sigma and the values recorded in its elements". These summaries are used both to generate a description of the app before the test-case driver starts and to report results as it executes.

4. CONCLUSION

We conclude by answering the questions posed in the ERA call for submissions. The *new idea* in our work is to generate documentation for Android apps from their formal specs rather than the code. And to generate documentation for the test cases based on the execution of the test case; the documentation identifies any violations of the expected behavior. This is achieved mostly by exploiting the facilities provided by the widely used Robolectric tool rather than by building a new tool. To our knowledge, analogous work has not been done, certainly not for mobile software. The two most related papers are the ones by Moreno *et al.* [8] and by Kamimura and Murphy [6]. Our on-going work on formalizing Android apps serves as the basis of the work reported in the current paper although the same approach can be used with any other formalization of Android. The feedback we desire from the ICPC gathering are general comments on the usefulness of documentation of the kind we propose generating; and, in particular, comments about what may be most useful to include and what may be omitted, in the interest of brevity, from the generated documentation.

5. REFERENCES

[1] Android. Monkey UI/Application exerciser. http://goo.gl/DkzXSd, 2010.

[2] Android. The android mobile platform. http://developer.android.com/about, 2012.

[3] Y Cheon, G Leavens. A simple approach to unit testing: JML and JUnit. *ECOOP*, pp. 231–255, 2002.

[4] N Dragan, M Collard, J Maletic. Reverse eng. method stereotypes. *ICSM*, pp. 24–34. IEEE CS, 2006.

[5] N Dragan, M Collard, J Maletic. Automatic identification of class stereotypes. *Int. Conf. Aut. Softw. Eng.*, pp. 1–10. IEEE CS, 2010.

[6] M Kamimura G Murphy. Generating human-oriented summaries of test cases. *ICPC*, pp. 215–218, 2013.

[7] Pivotal Labs. Robolectric 2.2 documentation. http://goo.gl/elaeWp, 2013.

[8] L Moreno, J Aponte, G Sridhara, A Marcus, L Pollock, K Shanker. Automatic generation of nat. lang. summ. for Java classes. *ICPC*, pp 23–32, 2013.

[9] B Philips and B Hardy. *Android*. Nerd Ranch, 2013.

[10] N Soundarajan, S Joshi, and Y Xu. Behavioral spec. of Android apps. `http://goo.gl/TNkFoC`, 2014.

[11] H van der Merwe, B van der Merwe, and W Visser. Verif. Android apps using JPF. *SEN*, 37(6):1–5, 2012.