An event calculus-based approach to reasoning about mobile apps

Swaroop Joshi, Yan Xu, Neelam Soundarajan
Computer Science & Engineering, Ohio State University
e-mail: {joshis, xuyan, neelam}@cse.ohio-state.edu

October 2013

Abstract

Software running on mobile platforms has become increasingly important. Given the essential tasks that many of these "apps" are intended to carry out, the importance of specifying their behavior precisely and adequately testing them against the specifications is clear. In this report, we describe our efforts to develop an approach to specifying and testing of apps for the Android platform. We explore the use of event-calculus for this, and discuss its pros and cons.

1 Introduction

The number and importance of applications running on mobile platforms has exploded in the last few years. From banking to e-commerce, from location-based targeted advertising to even enabling social revolutions, the phrase "there is an app for that" has gone from being an inside joke to a fact of life. Given the essential tasks that many apps are intended to carry out, we believe it is essential to develop suitable methods for specifying their behaviors precisely, as well as tools and techniques for testing the apps against their specifications. In this paper, we describe our efforts towards using event-calculus [18] to do this for apps for the Android platform.

As we will see, our approach focuses on the sequences of events that the app might engage in. This is appropriate given the central role that events, including such events as rotation of the device that are unique to mobile systems, play in mobile apps. The next section gives a brief background of the Android platform, current testing approaches for its apps, and event calculus. In Section 2, we explain our approach by applying it to `ZeroWins`, a simple one-activity game app, and `GeoQuiz`, a simple two-activity app. In Section 4, we turn to testing using these specs. Section 5 discusses the advantages and limitations of using event-calculus based approach to specifications.

2 Background

2.1 Android

In past few years, smartphones have evolved from being the devices to make communication with other similar devices through phone call or text message to being electronic devices with commuting power comparable to full-scale computers, running variety of software, and nearly replacing traditional desktop or laptops for certain utilities. Users can download and install the softwares ('apps') from manufacturers of the mobile devices, as well as from third-party developers through software stores like Google Play Store [3], Apple App Store [2] etc. Android, Google's open source platform, has become quite popular among app developers and hence the users. It commands 64% market share in global smartphone market, with 48 billion apps having installed from the Google Play Store [20].

Android runs on Linux-based kernel, with C-based middleware, libraries and APIs. Its application software runs on an application framework with Java-compatible libraries, which use the Dalvik virtual machine (DVM), adopted for Android. Most Android apps are written to work on DVM, but a developer can write a C/ C++ app and directly run it.

An android app has four basic components: `Activity, Service, ContentProvider` and `Broadcast Receiver`. An `Activity` is a unit of user interaction as well as execution. Each activity is separate from other activities. Typically, methods of one activity are not directly called from the code of another activity. Communication between two activities is carried through another class called `Intent`, and is handled by the OS. The `Service` class supports background functions. The `ContentProvider` class provides access to a data store for multiple applications, and the `Broadcast Receiver` allows multiple parties to listen for intents broadcast by applications [13].

A feature of Android which distinguishes it from most other Java apps is its event-driven architecture. An Android app does not have a single entry point. An Android program can be started in different places, depending on the user's previous position in the system, and her next intended actions. Unlike the hierarchical view of a traditional software (with `main()` calling a method `f()`, which in turn calls a method `g()` and so on), an Android app consists of a group of components which may invoke one another.

The *control-flow* of an android app can also be affected by the method calls reflecting the component life cycles. Figure 1 ([1]) shows the important stages and corresponding method calls in an `Activity`'s life cycle. Many of these method call backs are implemented by the `Activity` parent class and the programmer need not `@Override` those, but some of them, like `onCreate()` have to be implemented. Implementation of this method typically includes some basic application startup logic that should happen only once for the entire life of the activity, like loading the UI.

As Android is a platform for devices with touch-screens, which serve as the interface for input as well as output in most circumstances, the design of an android app is GUI-centric. It is based on the Model-View-Controller Framework (MVC). As per MVC, an object in the software has to be a model object, a view object or a controller object. A model object typically represents the data and current state of your app, a controller recognizes the input and updates the model, and this update in the model in turn

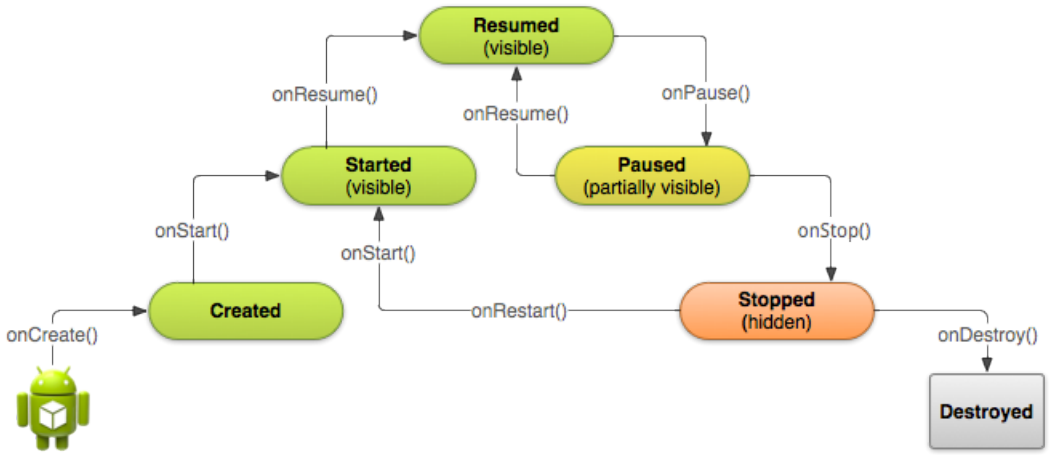


Figure 1: Some important stages and method call backs in Activity Life Cycle

modifies the view which is displayed on the screen. Figure 2 from [13, p.168] succinctly captures this concept. In android, View is typically the GUI loaded from XML files which describe the layout, Model is the application logic, and Controller is made up of various callbacks on the GUI-elements, like the `onClickListener()` on a button.

2.2 Android Testing

Several native and third party tools are available for testing an android app. Monkey [5], which comes with the Android SDK, is based on the infinite monkey theorem. The theorem states that a monkey hitting keys at random on a typewriter keyboard for an infinite amount of time will almost surely type a given text, such as the complete works of William Shakespeare. This android testing tool creates pseudo-random sequences of events that it feeds to the app. The goal is to see if the app crashes. Another tool Monkeyrunner [6], unrelated to Monkey despite the name, executes a python script which allows sending some specified input (data and event) to the app. Both the tools are fairly limited in their abilities, and it is not possible to test the *behavior* of an app, other than that it does not crash.

An app written is Java can of course be tested using JUnit [4]. Android SDK currently supports JUnit-3. Its limitation is, it cannot test for android-specific features, like lifecycle events. Another popular testing tool is Robolectric [7] which is built on top of JUnit. It intercepts the loading of Android classes and rewrites the method bodies [14]. It re-defines Android methods so they return default values, like `null`, `0`, or `false`, which can be checked in `assert` clauses to verify the behavior of the app at that point.

Orthogonal to these tools, there has been work on understanding the android-specific features of the apps and employing testing strategies targeted at those. A static analyzer which can handle features such as the event-based library and dynamic

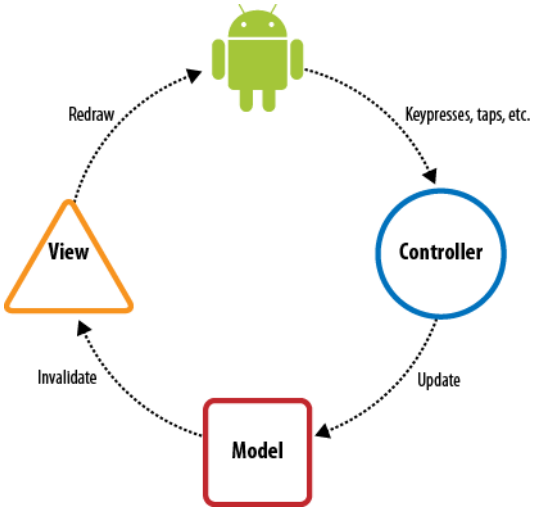


Figure 2: Model-View-Controller Concept

inflation of graphical views from declarative XML layout files was introduced in [16]. Converting the Dalvik byte code to Jimple and then performing static analysis on it was discussed in [10]. These works did not really target the crucial android-specific features of the app. A bug study and categorization of Android-specific bugs, which shows that many bugs are platform specific, and different from traditional bugs that occur in desktop applications was presented in [12]. They also present an event-generation based GUI testing approach. Another approach at GUI testing was presented in [9, 8, **?**] where GUI crawling and GUI ripping techniques are employed on android apps. Apart from this, verification of android apps through symbolic execution has also been investigated [19].

Although all of this research has improved the way testing of an android app is performed, we believe it has not been able to capture the key features of android framework stated above (non hierarchical structure, strong correlation with MVC, and dependence on life cycle events). Our work focuses on these aspects in order to provide a system targeted at android-like frameworks.

2.3 Event Calculus

Event Calculus (EC) was originally introduced for representing and reasoning about events and their effects on the state. Although it has been mainly used in Artificial Intelligence applications, it is also appropriate for reasoning about program behavior, especially for event-based systems; e.g., [11]. A key idea of EC is that a *fluent*, which is a property of the world that may change with time, is true at certain time-points if it has been *initiated* by an appropriate *event* (also called *action*) occurrence at some earlier time-point (or was true at system initiation time) and not *terminated* by another event occurrence since then; similarly, a fluent is false at certain time-points if it has

been *terminated* by an appropriate event occurrence at some earlier time-point (or was false at system initiation time) and not *initiated* by another event occurrence since then. Unlike temporal logics, EC does not introduce a new modal logic; instead, it is based on first-order predicate calculus, using suitable predicates and functions to express information about what happens when, etc. In addition, a set of axioms is used to constrain the set of models appropriately; these axioms will, in general be different for different application areas. The calculus we use is based on the complete event calculus described in [18]. The predicates are:

- $Initiates(a, b, t)$: if event a occurs at time t, then the fluent b holds at time t.
- $Terminates(a, b, t)$: if event a occurs at time t, then the fluent b does not hold at time t.
- $Releases(a, b, t)$: fluent b is not subject to the common sense law of inertia after event a at time t.
- $Initially_P(b)$: b holds at time-point 0.
- $Initially_N(b)$: b does not hold at time-point 0.
- $Happens(a, t)$: event a happens at t.
- $HoldsAt(b, t)$: fluent b holds at t.

EC enables us to infer a *goal* ($HoldsAt$ clauses) given a *narrative*, i.e., what events occur at various times ($Happens$, $Initially_P$, $Initially_N$ clauses, and temporal ordering), and *rules*, i.e., what their effects are ($Initates$, $Terminates$ clauses). This is called deduction. EC mechanism can also perform *abductive* reasoning, i.e., given a *goal* and *rules*, it can produce a possible *narrative*. This is equivalent to planning, because what we get is a bunch of actions and the order in which they should happen so that the goal is achieved under the given rules.

Several details can be added to the basic Event Calculus described above. Generally two types of events can be considered: $Perform(a, t)$ is used to indicate events performed by a user or an agent, while $Occurred(a, t)$ is used to designate events that occur as part of the system's response to the user actions or some other changes in the system. We use the terminology *actions* and *reactions* to identify these two types respectively. Now the $Happens$ clause can be re-written as

$$Happens(a, t) \equiv Perform(a, t) \vee Occurred(a, t)$$

Certain preconditions can be put on actions. For example, to indicate that an action a cannot be performed at time t, a predicate $Impossible(a, t)$ can be used. We re-write the $Happens$ predicate to incorporate this modification:

$$Happens(a, t) \equiv (Perform(a, t) \wedge \neg Impossible(a, t)) \vee Occurred(a, t)$$

3 Specifying Android Apps

A well designed app should dissolve into its components, which can invoke one another when needed. Not only that, but an activity from one app can directly invoke an

activity from another app. Thus, an activity is the basic unit of app execution, and that should be reflected in specification as well.

While an activity is interacting with the user through its GUI, there can be several other things happening in the background, like an incoming phone call, change in the devices GPS location, battery going below a critical threshold, or the device being rotated, etc., which the OS has to handle. Some of these events may affect the current activity as well. In most of the cases, the current activity will respond to these system-events in the default way. Due to this reason we adopt a two-tier approach for activity specification: our spec will assume the default behavior of the activity with respect to such system-events, unless explicitly mentioned in the spec. For example, a simple game app will not respond to the battery going critically low, but an app handling bank transactions may first want to save all its current data on the server. In that case, the spec of the bank app should specify how it handles the system-event of battery going low.

The event-driven nature of android apps prompted us in searching for a possibility of using EC to describe the behavior of android apps. Upon investigating, we realized that there is an even closer relation between the two frameworks.

3.1 Relation between MVC and EC constructs

As we have seen in Section 2.1, design of an android app is based on the Model-View-Controller framework. Event Calculus, in a sense, supports this framework. View of an android system consists of all the elements of the GUI of an activity, built from the XML files in the layout. Most of the user input and output takes place through this GUI which is displayed on the touch-screen of the device. Thus, both *actions*, i.e., what user wants to do with the app, and *reactions*, i.e., how the activity responds to those actions, are related to the View. Model, on the other hand, consists of the data which represents the current state of the activity. This is similar to the set of *fluents* in the event calculus. Controller, which is made up of various callbacks on the GUI-elements, can be represented by *action-effects*, i.e., *Initiates, Terminates* clauses in the EC. This relation simplifies identifying events and fluents for specifying an activity. Let us have a look at following two examples.

3.2 Example: `ZeroWins` and `GeoQuiz`

Let us take a detailed look at how we designed the spec for `ZeroWins` and `GeoQuiz` apps.

3.2.1 EC based specs for `ZeroWins`

`ZeroWins` is a simple two-player game. It allows two users, $p0$, $p1$ to play a *number guessing game* with $p0$ and $p1$ taking alternate turns. In its internal state, the app keeps a value which is the sum of all the numbers that $p0$, $p1$ have typed into the system thus far during their turns. In order to win, a user, say, $p1$, when it is her turn, must guess and type in a number that when added to the current sum gives zero in which case $p1$ is the winner and the game ends. Before $p1$ types in her guess, she can

ask for help; she can give a number k and ask, once per turn, whether, in order to win, she should choose a number k' such that k' is less than k or k' is greater than or equal to k and the system will respond (truthfully!). Of course, as each user types in her next guess during her turn, it is hidden from the other user. When `ZeroWins` starts, it sets its internal value to a randomly generated integer number between 0 and 100.

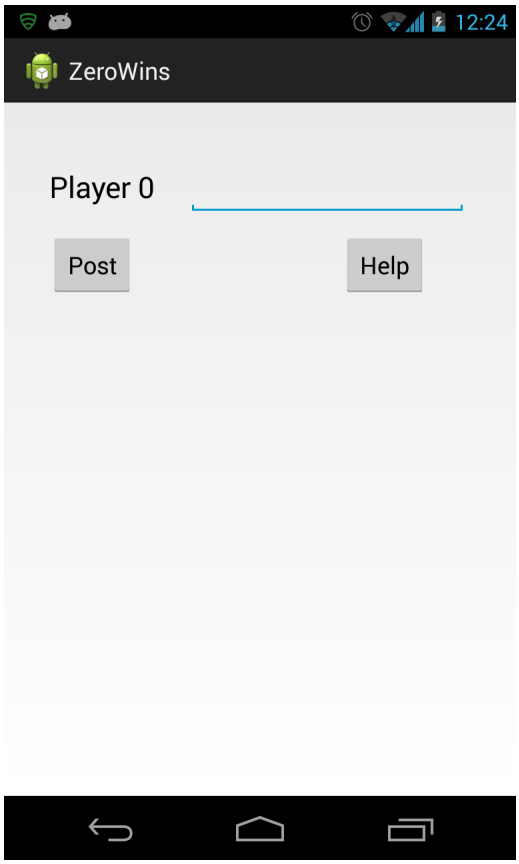


Figure 3: `ZeroWins` GUI

The game is implemented using a single activity whose GUI is shown in Figure 3, and Figure 4 represents its MVC. From this, and our discussion on relation between MVC and EC, we can identify two possible *actions* of this activity: `Post` and `Help`, corresponding to the two buttons. Inputting the number through the `EditText` widget provided can also be an action, but the way it is handled is not different from the default handling of any text input, so we need not specify that in out spec. The fourth element in the View is the TextView which displays the player number for the player who is supposed to go next, and that is a *reaction* from the activity. Fluents are given by the model; in this case there are essentially two variables which represent the state of the activity. So our fluents will be $sum(x)$ and $player(p)$. There are two more

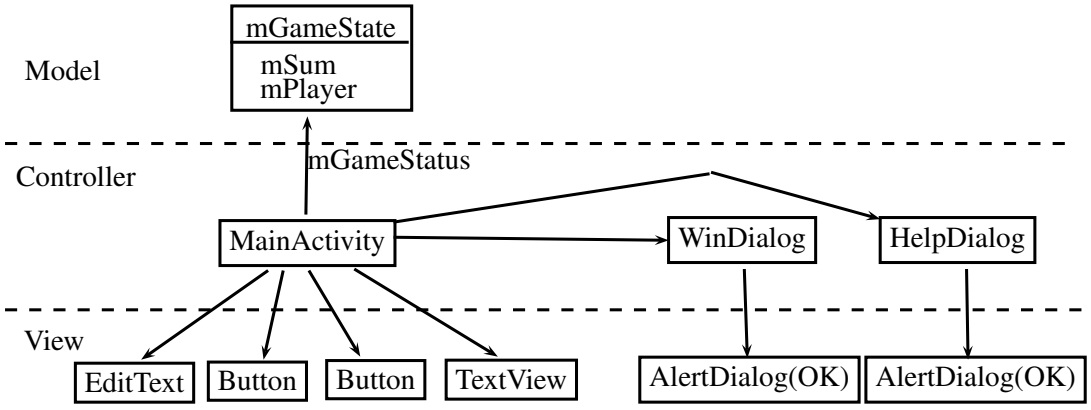


Figure 4: `ZeroWins` MVC diagram

elements in the View: the two dialog boxes which show up when a player wins, or if a player asks for help. These, too, are reactions. Each of these dialogs expect the user to click the OK-button to resume the activity; these are again actions.

Thus, our EC model for `ZeroWins` app's sole activity, `MainActivity` is:

Actions: Post(y), Help(y), WinDialogOk, HelpDialogOk

Reactions: ShowPlayer(p), ShowWinDialog(msg), ShowHelpDialog(msg)

Fluents: Sum(x), Player(p)

We introduce some new constructs based on our understanding of the app behavior – *PrevState* which represents a previous state of the activity:

$$PrevState([b_1, b_2, \ldots, b_n], t) \equiv HoldsAt(b_i, t') \wedge \neg Clipped(t', b_i, t) \wedge t' < t, \quad \forall i. 1 \leq i \leq n$$

Happening of an event can *trigger* a reaction:

$$Triggers(a_1, a_2, t) \equiv Occurs(a_2, t) \leftarrow Happens(a_1, t)$$

We also introduce some syntactic sugar to simplify our specs:

$$InitiatesAll(a, [b_1, b_2, \ldots, b_n], t) \equiv Initiates(a, b_i, t) \qquad \forall i. 1 \leq i \leq n$$

$$TerminatesAll(a, [b_1, b_2, \ldots, b_n], t) \equiv Terminates(a, b_i, t) \qquad \forall i. 1 \leq i \leq n$$

Now let us discuss the *action effects*:

Action effects - Post(y):

- $InitiatesAll(Post(y), [Player(p), Sum(x)], t)$
  $\leftarrow PrevState([Player(p'), Sum(x')], t) \wedge p \neq p' \wedge p \in 0, 1 \wedge x = x' + y \wedge x \neq 0$
- $Triggers(Post(y), ShowWinDialog(p'), t) \leftarrow PrevState([Player(p'), Sum(x')], t) \wedge$
  $x = x' + y \wedge x = 0$
- $TerminatesAll(Post(y), [Player(p'), Sum(x')], t) \leftarrow PrevState([Player(p'), Sum(x')], t)$

Action effects - Help(y):

- $Triggers(Help(y), ShowHelpDialog(\text{“}Guess{+}Sum \geq 0\text{”}), t) \leftarrow PrevState([Sum(x')], t) \wedge x' + y \geq 0$
- $Triggers(Help(y), ShowHelpDialog(\text{“}Guess{+}Sum < 0\text{”}), t) \leftarrow PrevState([Sum(x')], t) \wedge x' + y < 0$

Action effects - WinDialogOk: Clicking the ok button on the Win Dialog does not change any fluents.

Action effects - HelpDialogOk: Clicking the ok button on the Help Dialog does not change any fluents.

Effects of lifecycle events: Certain activity life cycle events (Figure 1) may also affect the activity under running state.

Starting the activity loads a new random number in *Sum* and the *Player* is set to 0. This is the initial state. If a previous state exists, that is loaded.

- $InitiatesAll(Start(ZeroWinsMain), [Player(0), Sum(x)], t) \leftarrow x \textit{ is a random integer} \in (0, 100)$
- $InitiatesAll(Start(ZeroWinsMain), [Player(p), Sum(x)], t)$
  $\leftarrow PrevState([Player(p), Sum(x)], t1) \wedge t1 < t$

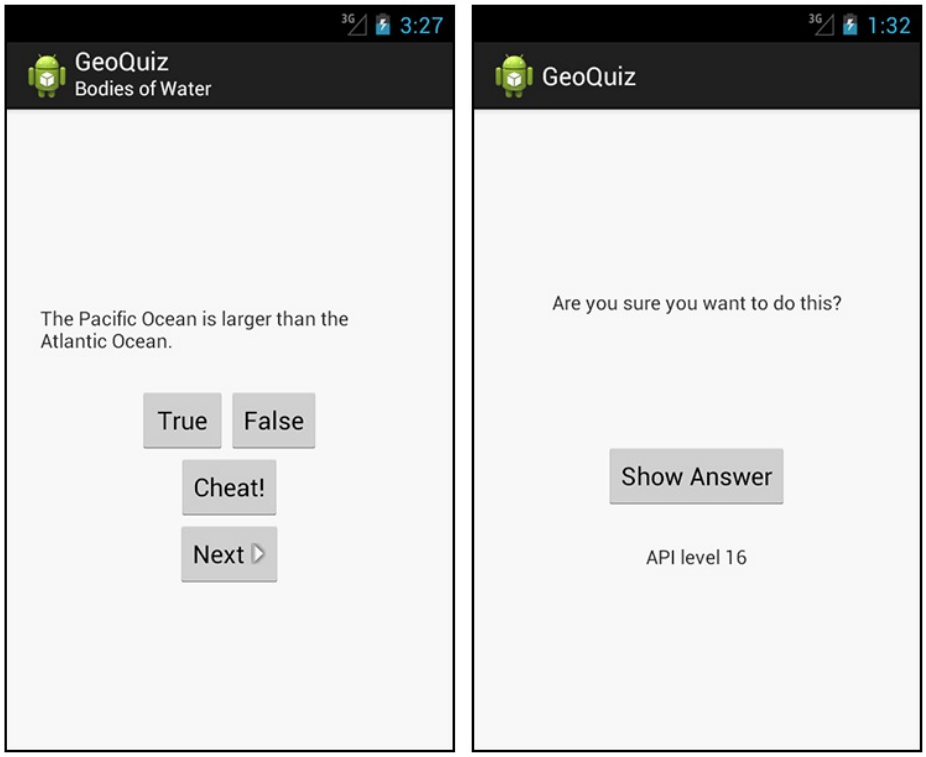
Pause and *Stop* save the current state. Since we have captured this in *PrevState* already, no action effects are explicitly specified. Finally, all fluents are terminated when the activity is destroyed.

- $TerminatesAll(Destroy(ZeroWinsMain), [Player(p), Sum(x)], t)$

3.2.2 EC based specs for `GeoQuiz`

`GeoQuiz` is a simple two-activity app discussed in the initial chapters of [17]. Its main activity `QuizActivity` displays a series of geography questions which can be answered True or False. It provides two buttons (`True, False`) to record the answer. It also provides a `Next` button to go to the next question in the sequence, and a `Cheat` button to launch the `CheatActivity` which gives the user an option to check the answer, and reports back to `QuizActivity` if the user did see the answer. Figure 5 shows the GUI of these two activities.

Based on our previous discussion on the relation between MVC and EC constructs, the user actions of `QuizActivity` activity are the button presses of the four buttons visible. One activity-reaction is the display of appropriate question. Another reaction which cannot be inferred from the GUI is a *Toast* message, which is the feedback of the activity when the user records her answer. The fluents are based on the current state variables. In the case of `QuizActivity`, they are `Current(q)` indicating q is the index of current question, and `Cheater` indicating the user has cheated on this question. For `CheatActivity`, there is only one action, `Show` which corresponds with clicking the “Show Answer” button, and a reaction `ShowAns(ans)` which corresponds with displaying the `ans` on the `TextView`. We also require the knowledge of whether the



(a) `QuizActivity` (b) `CheatActivity`

Figure 5: GeoQuiz GUI

answer to the current question is `true` or `false`. This is not a fluent because its value does not change with time. We capture this information in a boolean `AnsIsTrue(q)`.

Actions in one activity cannot *happen* while the user is in some other activity. For this, we use the predicate *Impossible*, introduced in [15]. `Impossible(a, t)` indicates that event a cannot *happen* at time t. This is expressed as

$$Happens(a,t) \equiv (Perform(a,t) \wedge \neg Impossible(a,t)) \vee Occurred(a,t)$$

We introduce a syntactic sugar:

$$ImpossibleAll([a_1, \cdots, a_n], t) \equiv Impossible(a_i, t). \quad \forall a_i$$

So we have

$$ImpossibleAll([\texttt{True}, \texttt{False}, \texttt{Cheat}, \texttt{Next}, \texttt{ShowToast}(\texttt{msg})], t) \leftarrow HoldsAt(Running(\texttt{CheatActivity}), t)$$

and

$$ImpossibleAll([\texttt{Show}], t) \leftarrow HoldsAt(Running(\texttt{QuizActivity}), t)$$

Note that $Running(a), Stopped(a)$, etc. are system level fluents corresponding to the states in activity life-cycle, and $Resume(a), Pause(a)$, etc. are actions corresponding to method calls. Now the specs for `QuizActivity` can be written as follows:

Initial State:

- $Initiates(Start(QuizActivity), Current(0), t)$

Action effects - `True`:

- $Triggers(True, Toast(\text{“Cheating is wrong”}), t) \leftarrow HoldsAt(Cheater, t) \wedge HoldsAt(Current(q), t) \wedge AnsIsTrue(q)$
- $Triggers(True, Toast(\text{“Correct!”}), t) \leftarrow \neg HoldsAt(Cheater, t) \wedge HoldsAt(Current(q), t) \wedge AnsIsTrue(q)$
- $Triggers(True, Toast(\text{“You are bad at cheating”}), t) \leftarrow Holds([Cheater, Current(q)], t) \wedge \neg AnsIsTrue(q)$
- $Triggers(True, Toast(\text{“Incorrect!”}), t) \leftarrow \neg HoldsAt(Cheater, t) \wedge HoldsAt(Current(q), t) \wedge \neg AnsIsTrue(q)$

Action effects - `False`:

- $Triggers(False, Toast(\text{“Cheating is wrong”}), t) \leftarrow HoldsAt(Cheater, t) \wedge HoldsAt(Current(q), t) \wedge \neg AnsIsTrue(q)$
- $Triggers(False, Toast(\text{“Correct!”}), t) \leftarrow \neg HoldsAt(Cheater, t) \wedge HoldsAt(Current(q), t) \wedge \neg AnsIsTrue(q)$
- $Triggers(False, Toast(\text{“You are bad at cheating”}), t) \leftarrow Holds(Cheater, t) \wedge HoldsAt(Current(q), t) \wedge AnsIsTrue(q)$
- $Triggers(False, Toast(\text{“Incorrect!”}), t) \leftarrow \neg Holds(Cheater, t) \wedge HoldsAt(Current(q), t) \wedge AnsIsTrue(q)$

Actions effects - `Next`:

- $Initiates(Next, Current(p), t) \leftarrow HoldsAt(Current(q), t) \wedge p = (q + 1) \bmod N$, where N is the total number of questions.
- $TerminatesAll(Next, [Current(q), Cheater], t) \leftarrow HoldsAt(Current(q), t)$

Actions effects - `Cheat`:

- $Triggers(Cheat, Start(CheatActivity))$
- $Initiates(Cheat, BundleHas(\langle \text{“answer”}, AnsIsTrue(q)\rangle), t) \leftarrow HoldsAt(Current(q), t)$

Derived fluent - Cheater:

- This fluent gets its value from the element in the bundle set by `CheatActivity`. Thus, $HoldsAt(Cheater, t) \leftarrow HoldsAt(BundleHas(\langle \text{“has cheated”}, \texttt{true}\rangle), t)$

Now let's look at the action effects of the `CheatActivity`.

Action effects - `Show`:

- $Initiates(Show, BundleHas(\langle \text{“has cheated”}, \texttt{true}\rangle), t)$
- $Occurs(ShowAns(\text{“True”}), t) \leftarrow HoldsAt(BundleHas(\langle \text{“answer”}, \texttt{true}\rangle), t)$
- $Occurs(ShowAns(\text{“False”}), t) \leftarrow HoldsAt(BundleHas(\langle \text{“answer”}, \texttt{false}\rangle), t)$

4 Testing against the spec

Specifying an app is one part of the behavior based testing process. Testing the actual behavior of the software against the spec is the other part. While the spec is written in EC, the app is written in Java. Therefore a testing approach requires establishing the relation between the two. As discussed in Section 2.2, approaches like Monkey and Monkeyrunner are not well suited for testing the behavior of an app. Robolectric, however, can be used to extract the widgets and performs the actions using them just like a user would do while using the app. These actions comprise a *narrative* which can be mapped to the *Happens* clauses in the corresponding EC model. Current state of an EC model can be obtained by looking at the *HoldsAt* clauses on fluents, which are booleans representing the variables in the system. In a Robolectric test case, these can be mapped to `assert` statements, inserted at various places in the code; if a certain fluent should *hold* at a particular time-point in the EC narrative, the corresponding `assert` should hold at the corresponding time-point in the test case narrative.

4.1 Creating narratives manually

By looking at the app specification, a tester can manually come up with some narratives and test them by using `assert` statements as described above. For example, for the `GeoQuiz` activity, this simple narrative is shown in Listing 1, from which a test case as shown in Listing 2, can be created.

```
HoldsAt(Running(QuizActivity), t0)
Happens(TrueButon, t1)
HoldsAt(Correct, t1)
Happens(Next, t2)
HoldsAt(Current(2), t2)
Happens(Cheat, t3)
HoldsAt(Running(CheatActivity), t3)
Happens(Back, t4)
HoldsAt(Running(QuizActivity), t4)
t0 < t1 < t2 < t3 < t4
```

Listing 1: An EC Narrative for `GeoQuiz`

```
public void testNarrative1() throws Exception {
solo.assertCurrentActivity("Check on first activity",
  QuizActivity.class);
solo.clickOnButton("True");
assertTrue(solo.searchText("(?i).*Correct.*"));

solo.clickOnButton("Next");
String questionText = solo.getString(
  com.bignerdranch.android.geoquiz.R.string.question_2);
final String q2 =
  "The Suez Canal connects the Red Sea and the Indian Ocean.";
assert (questionText.equals(q2));

solo.clickOnButton("Cheat!");
solo.assertCurrentActivity("Check on the next activity",
  CheatActivity.class);
Activity cheatActivity = solo.getCurrentActivity();
Solo solo2 = new Solo(getInstrumentation(), cheatActivity);
solo2.clickOnButton("Show Answer");

solo2.goBack();
solo.assertCurrentActivity("Back to QuizActivity",
  QuizActivity.class);
}
```

Listing 2: Robolectric test case of the EC narrative for `GeoQuiz`

4.2 Generating narratives automatically using planning

Of course, creating such narratives manually is a tedious job and may not cover all possible test scenarios. We propose a better approach which is possible due to the use of EC model is using planning for generating narratives. Given a well formed

spec (action effects or *rules*) and a certain *goal* (*HoldsAt* clauses), planning algorithm can come up with all possible *narratives* (*Happens* and temporal ordering) which can achieve that goal. In fact, using Prolog, this can be done automatically. Robolectric test cases can be created by translating those narratives and checked for appropriate `assert` clauses place appropriately.

5 Limitations of EC approach

Event calculus based formalism provides a good candidate for specification of android applications due to their event driven design. The action–fluent–action effect of EC match the model–view–controller design of android apps. EC based formalisms are basically first-order logic formulas and hence they can be easily tested using established tools like a prolog meta-interpreter. This simplicity, unfortunately, presents two limitations as well. One, the specs become very lengthy, and two, they have to be written in a way the meta-interpreter accepts them, more specifically, in terms of horn clauses. Moreover, for a complex activity, reading the specs can become cumbersome, and keeping the specifications bug-free itself could be a non-trivial task. We also feel that representing the complex concepts of activity life cycles, *unpredictable* sequence of events that can take place in an app, and numerous possibilities of interaction with the rest of the system require a more sophisticated formalism for specification of android apps.

6 Conclusion

Highly event driven nature of android apps requires a modeling framework that can capture those events and their effects. Event calculus is a good candidate for this, and it has been used for reasoning of softwares. We have been able to produce the EC based specifications of some sample apps, and a strategy to automatically test the app against those spec. Complicated nature of the android platform, however, poses challenges in terms of scaling this approach to advanced apps. Some other framework which can model the unique features of android platform will be more suitable to specify, and then test, the behavior of these apps.

References

[1] Android developer documentation. http://developer.android.com/index.html.

[2] Apple play store. www.appstore.com.

[3] Google play store. https://play.google.com/store.

[4] Junit – a programmer-oriented testing framework for java. http://junit.org.

[5] Monkey. http://developer.android.com/tools/help/monkey.html.

[6] Monkeyrunner. http://developer.android.com/tools/help/monkeyrunner_concepts.html.

[7] Robolectric. www.robolectric.org.

[8] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. A gui crawling-based technique for android mobile application testing. *Software Testing Verification and Validation Workshop, IEEE International Conference on*, 0:252–261, 2011.

[9] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Gennaro Imparato. A toolset for gui testing of android applications. *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, 0:650–653, 2012.

[10] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. Using gui ripping for automated testing of android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 258–261, New York, NY, USA, 2012. ACM.

[11] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. Dexpler: converting android dalvik bytecode to jimple for static analysis with soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, SOAP '12, pages 27–38, New York, NY, USA, 2012. ACM.

[12] Dimitra Giannakopoulou and Jeff Magee. Fluent model checking for event-based systems. In *In Proceedings of FSE*, pages 257–266. ACM Press, 2003.

[13] Cuixiong Hu and Iulian Neamtiu. A gui bug finding framework for android applications. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 1490–1491, New York, NY, USA, 2011. ACM.

[14] Z. Mednieks, L. Dornin, G.B. Meike, and M. Nakamura. *Programming Android*. O'Reilly Media, 2011.

[15] D.T. Milano. *Android Application Testing Guide*. Community experience distilled. PACKT PUB, 2011.

[16] Rob Miller and Murray Shanahan. Some alternative formulations of the event calculus. In Antonis C. Kakas and Fariba Sadri, editors, *Computational Logic: Logic Programming and Beyond*, volume 2408 of *Lecture Notes in Computer Science*, pages 452–490. Springer, 2002.

[17] Étienne Payet and Fausto Spoto. Static analysis of android programs. *Information and Software Technology*, 54(11):1192 – 1201, 2012.

[18] B. Phillips and B. Hardy. *Android Programming: The Big Nerd Ranch Guide*. Big Nerd Ranch Guides. Pearson Education, Limited, 2013.

[19] Murray Shanahan. The event calculus explained. In *Artificial Intelligence Today*, pages 409–430. 1999.

[20] Heila van der Merwe, Brink van der Merwe, and Willem Visser. Verifying android applications using java pathfinder. *SIGSOFT Softw. Eng. Notes*, 37(6):1–5, November 2012.

[21] Wikipedia. Android (operating system) — Wikipedia, the free encyclopedia, 2013. [Online; accessed 12-October-2013].