Module 0: Introduction to Discrete Modeling in Android

1 Summary

This module's goal is to familiarize students with modeling Android applications using Finite State Machines (FSMs). Students will learn and observe the importance of modeling in application design. They will also get introduced to the basics of creating simple Android applications based on the FSM models.

2 Hardware Requirements

Any Android compatible device with a touch screen.

3 Additional References

- Chapter 3 in "Introduction to Embedded Systems, A Cyber-Physical Systems Approach" by Lee and Seshia. http://LeeSeshia.org, ISBN 978-0-557-70857-4, 2011.
- "Finite State Machines and Modal Models in Ptolemy II" by Edward Lee. *http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-151.pdf*
- Android Developer's Guide. *http://developer.android.com/guide/index.html*

4 Discrete Modeling in Android

Finite State Machines (FSMs) are a modeling tool useful in representing discrete things, such as programs. They consist of a set of states and a set of transitions between those states. Usually, only one state is labeled as the initial state, where execution of the program begins. As the program executes, the finite state machine model represents the state of the program, taking transitions to other states based on certain types of inputs to the program. The program finish can be modeled as a final state in the diagram, but many Android apps may run continously, and their model would not have a final state. FSM modeling is commonly a design-time activity, which means it is done prior to the majority of the implementation work. In other words, we use FSMs to capture the program states, and how they change based on the inputs

The benefit of modeling using FSMs is that the resulting software can be more robust, and most of the corner cases will already have been thought our and represented in the model. We want to design FSMs that have the property of *receptiveness*, where each state contains a transition for all possible inputs. This ensures that we handle all corner cases properly, and that the model has considered all possible inputs in all software states.

4.1 Application 0: Simplest Android Security Device

To start, let's design a security access Android application that requires the user to enter a particular security code to unlock the device. The security code is entered as a 3 digit sequence of button presses. If the correct sequence is entered then the device unlocks, while an incorrect sequence results in the user being asked to

enter the sequence again. For simplicity, assume that the 3 digit sequence is hard coded (i.e. there is no way for the user to enter their own sequence for now).

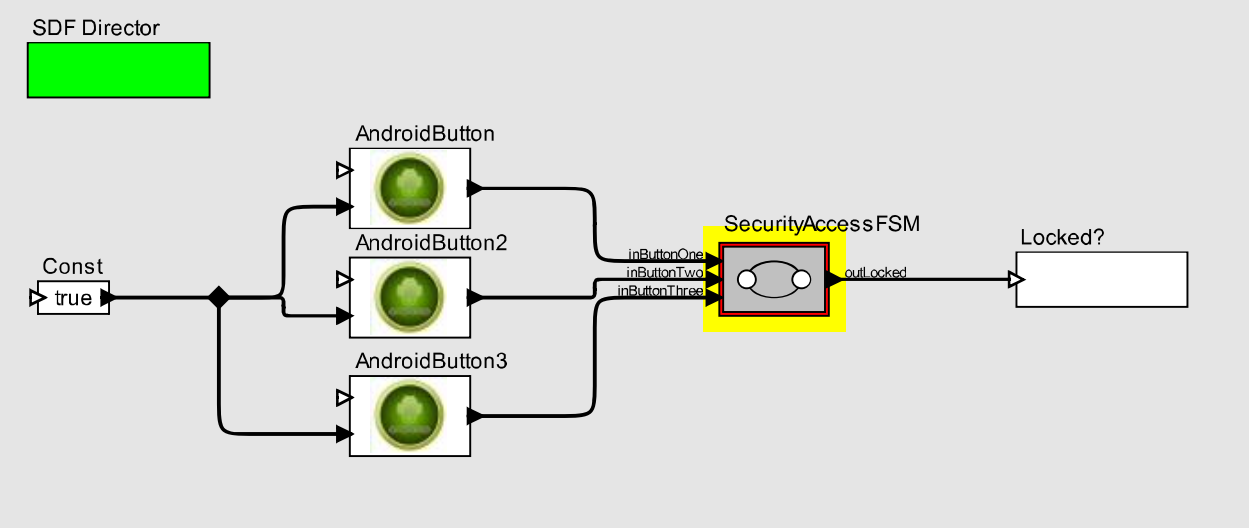
4.1.1 Starter PtolemyII Model

Most of the modeling in this sequence of course modules is done using Ptolemy II. This modeling environment matches cyber-physical systems applications quite well in that it is supports hybrid modeling, where discrete models can be easily coupled with continuous ones. You should download Ptolemy II from (here), and install it. We also provide some customized Ptolemy actors that will make modeling Android applications a bit easier. Those should be downloaded (here).

To help focus on only the important aspects of modeling this application, download the starter Ptolemy model, which will set up some of the necessary Ptolemy actors. A starter model will be available for each application that we are working on. It contains a set of Ptolemy II actors that enable testing and simulation of the application.

We can think of a Ptolemy simulation as consisting of multiple hierarchical layers. The starter model will provide you with the highest layer, which describes the application's inputs (e.g. simulated Android buttons), outputs (e.g. Android output boxes), and processing components defined using one or more FSMs. The FSMs and their states and transitions belong at a lower layer of the hierarchy. They will be your responsibility! Each layer of a Ptolemy simulation contains one director, which describes how the actors communicate with each other. The simplest director is the Synchronous DataFlow (SDF) director, which basically executes each actor when it's inputs are ready.

By providing you with highest layer of the modeling hierarchy, we would like to (for now) direct your focus only on modeling of the processing (discrete aspects) of the application using a FSM. The FSM is going to be the part of the model that will be directly translated to the implementation code, so it makes sense for you to focus on that. Below is the starter model for the first application:



4.1.2 Interacting with Ptolemy II

Here are some extremently directions on how to interact with Ptolemy II. A Ptolemy II application can be executed by pushing the big "Play" button in the toolbar, at which point you will see the output pop up in the output *Locked?* box. It is also possible to slowly simulate the execution slowly (which can be very useful in debugging your FSM), by choosing a in longer interval (e.g. 1000ms) in the *Debug→Animate Execution* menu.

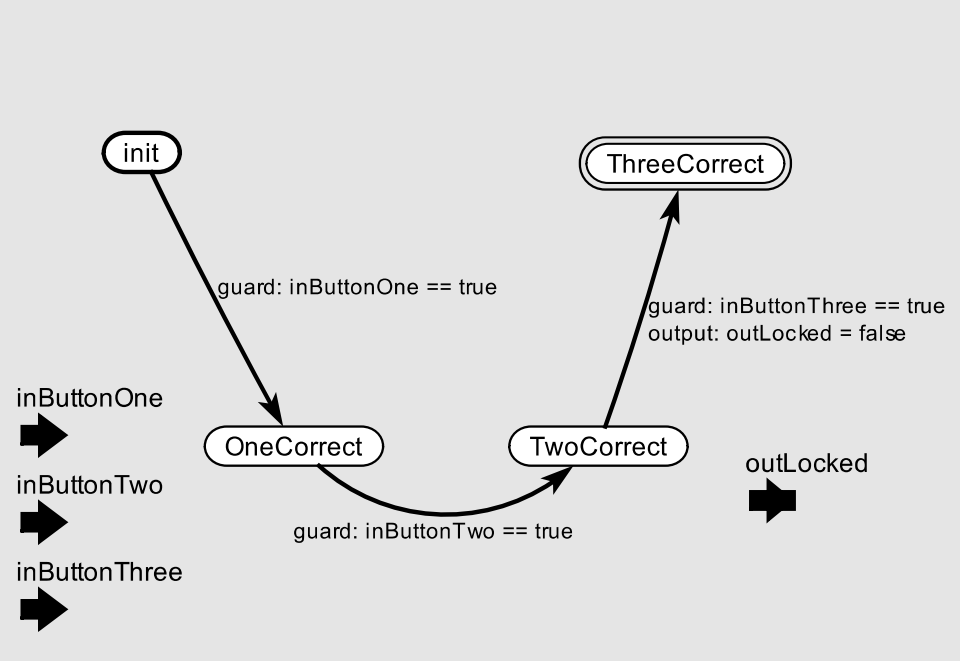
An important aspect of Ptolemy modeling is changing an Actor's parameters by double clicking on them. Each actor has specific parameters, so you have to use a bit of trial and error when modifying them. To edit

the FSM, which you will need to do in the next section, right click the FSM actor and select "Open Actor". A new screen will appear with the ability to define an FSM. To instantiate a state, you should drag the state icon from the left side pane on to the center of the screen. A transition between states can be drawn by holding the Ctrl key and clicking on a state. Each transition and state has parameter that you can adjust. A transition's parameters (*guardExpression, outputActions, and setActions*) are very important in drawing an FSM. The *guardExpression* defines a transition's guard and it contains a logical expression (expression that evaluates to true or false) (e.g. timer == 100, inButtonOne_isPresent). On the other hand, *outputActions* and *setActions* can be used to change the value of an output port or a variable. These changes occur when the *guardExpression* has evaluated to true.

Once you are finished with the FSM, pushing the "Play" button will run the entire model, buttons and all. As the model executes you can determine if your FSM (and your design) is correct before you begin focusing on implementing that design in Android. As you will see later, having a correct FSM makes the Android implementation very straightforward.

4.1.3 Modeling

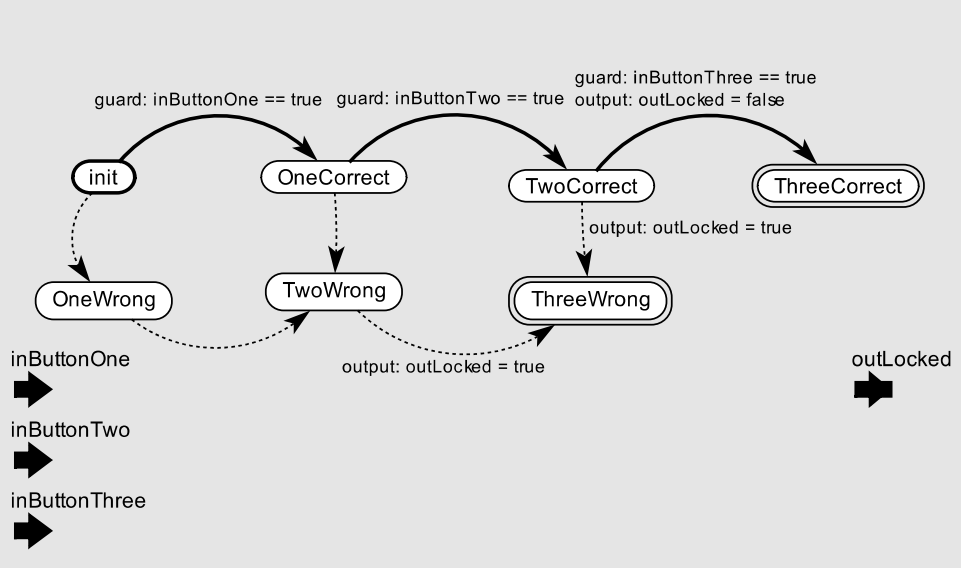
We begin by attempting to model the security access device using an FSM. This will hopefully help us think about what should be implemented in the Android app later.



This FSM has four states (`init`, `OneCorrect`, `TwoCorrect`, `ThreeCorrect`), three inputs corresponding to the three buttons (`inButtonOne`, `inButtonTwo`, `inButtonThree`) and one output (`outLocked`). The `init` state is the initial state - where execution begins - while the `ThreeCorrect` state is the final state. All of the inputs and outputs in this FSM are boolean – also referred to as pure. The transitions (arrows) between the states have guards and outputs. Guards have to become true for the transition to occur, and when they do the output statement is performed.

The FSM is good at handling correct inputs, but its obviously incomplete as it doesn't encode what happens if an incorrect button is pushed. An easy way to determine if the FSM is complete is see whether it handles each input in all states. For instance, when the application is in the `OneCorrect` state, only a transition on `inButtonTwo` is provided, while we don't know what to do if `inButtonThree` was pressed.

We enhance the our initial attempt at a FSM with the ability to handle incorrect inputs, in the following way:



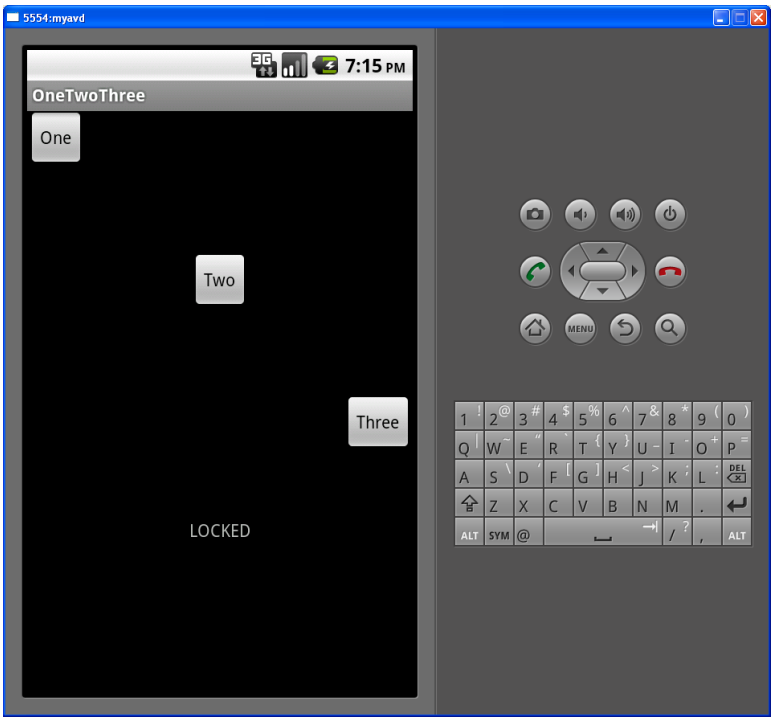
This FSM has default transitions (encoded by the dotted lines) that execute whenever none of the other transitions for the states can be taken. We also provide three new states: `OneWrong`, `TwoWrong` and `ThreeWrong`, intended to encode the fact that the user incorrectly entered one of the numbers in the secury code. After getting the wrong input, the system will still wait to recieve the full three digit sequence before informing the user that the entry was incorrect by issuing the signal `outLocked` that locks the device.

4.1.4 Implementation

In our discussion here, we assume a properly installed Android SDK. The SDK comes with simulator, which can be used, although an actual device is preferable

User Interface. Android application usually utilize one or more `Activities` that encode one UI "screen". Most Android applications instantiate one or more activities, which communicate via an `Intent` object. In this simple application we plan to have only one activity, which will contain the buttons and the output of our security access device. In Java code, an activity corresponds to a user-defined class that extends the Android specified `android.app.Activity` base class.

Android user interface elements that belong to an `Activity` can be drawn graphically, or encoded in an XML-based language. For our application, we need to provide three buttons, labeled as 1, 2, and 3. Use either the Android graphical UI creation tool or directly edit the activity's XML file (in the res/layout subfolder) to create these button, change their `Text` property to properly label them, and change their `Id` property so that we can recognize them in the activity's Java code. *We will not provide you with a starter model for the implementation part, since as computer scientists, we assume you can navigate this world much easier.*



Converting the FSM logic into Java. To easily implement things based on our FSM, we define an enumeration type that defines all of the different states (i.e. `OneCorrect`, `OneWrong`, `TwoCorrect`, `TwoWrong`, etc.). Also we define a global variable of this enumeration type to hold the current state that the application is in.

```
enum AppState {
      Init,
      OneCorrect,
      TwoCorrect,
      ThreeCorrect,
      OneWrong,
      TwoWrong,
      ThreeWrong
}

AppState status = AppState.Init;
```

To implement our application logic, we add three methods (which must take a `View` parameter and return `void`) to the activity class that will be called for each button push. For instance:

```
public void clickedButtonOne(View view)
{...}
public void clickedButtonTwo(View view)
{...}
public void clickedButtonThree(View view)
{...}
```

We must connect each button to its corresponding method from the above, which is accomplished by changing the `OnClick` property in the UI XML file. This property should contain the name of the method that will be called, so, for example, this property for the first button should contain *clickedButtonOne*. Below is the implementation of that method, extracted from the Ptolemy II FSM we completed. The key insight in performing this conversion is to encode all of the state transitions concerning that particular button, which often involves most or all of the states.

```
public void clickedButtonOne(View view) {
    TextView textBox = (TextView) findViewById(R.id.StatusText);
    if(status == AppState.Init)
    {
         status = AppState.OneCorrect;
    }
    else if(status == Appstate.OneCorrect)
    {
         status = AppState.TwoWrong;
    }
    else if(status == AppState.TwoCorrect)
    {
         status = AppState.ThreeWrong;
    }
    else if(status == AppState.OneWrong)
    {
         status = AppState.TwoWrong;
    }
    else if(status == AppState.TwoWrong)
    {
         textBox.setText("WRONG CODE! DEVICE LOCKED");
         status = AppState.ThreeWrong;
    }
    else if(status == AppState.ThreeCorrect || status == AppState.ThreeWrong)
    {
         //DO NOTHING
    }
}
```

Make sure you use "Organize Imports" in Eclipse (Ctrl+Shift+O) to automatically insert the necessary Java import statements for the Android classes

If you copy the code above, you have to ensure that the name following "R.id." matches the `Id` field in the activity's XML file. This is another way in which the Java code and the UI written in XML can interface.

In the clicked method of each button, such as `clickedButtonOne` for the first button, we provide code that implements the logic for that specific button click that directly corresponds to our FSM: for each of the button pushes, we provide code to encode all the state transitions. Since we are confident in the FSM model we built, this part should be fairly straightforward to accomplish, resulting is a **c**orrect and robust implementation.

4.1.5 Application Extensions

Students can perform the following simple enhancements to the initial security code application. They require a change to the FSM followed by a change in the implementation, but should generally be straightforward to complete.

- **Add a reset button.** A reset button, when pressed, resets the sequence entries that the user has made thus far. It puts the application in the initial state of expecting sequence input.
- **Add a change sequence button.** The change sequence button can be pressed only after the device is in the unlocked state. When pressed it allows for the next three digits to denote a new sequence. After the new sequence is entered the device becomes locked again.