Module 2: Discrete Time FSMs

1 Summary

This module's goal is to familiarize students with modeling Android applications using Timed Finite State Machines (FSMs). Students will learn and observe the importance of modeling in applications that utilize a discrete concept of time.

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*

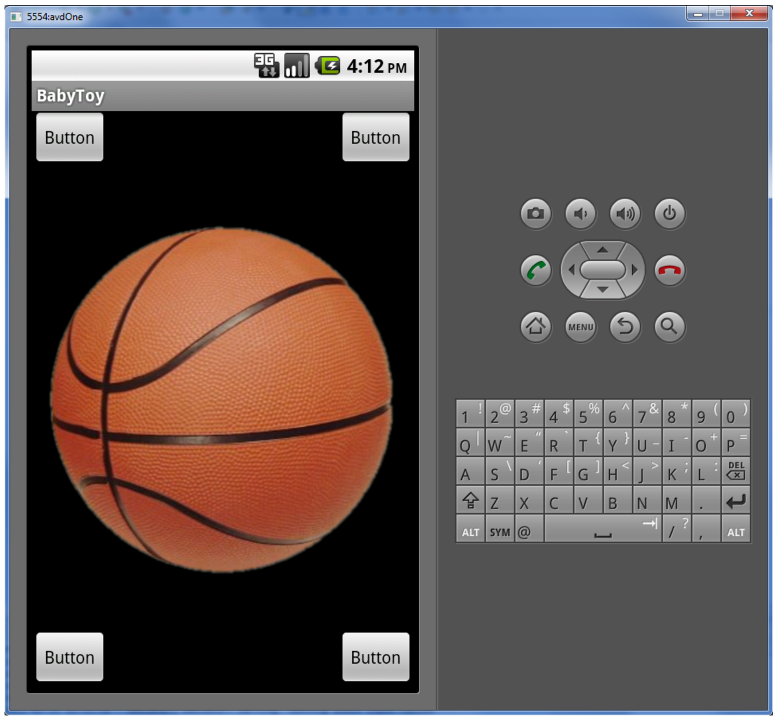
4 Discrete Modeling in Android

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.

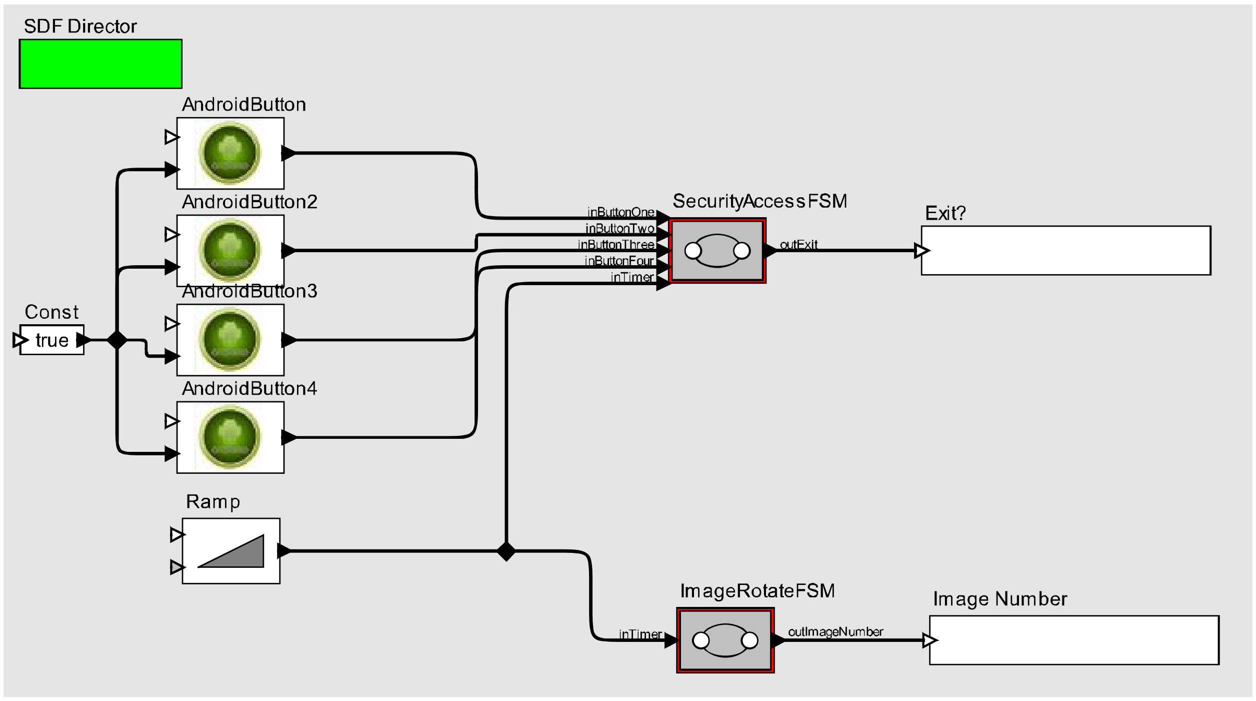
4.1 Application: Baby Toy

Students will design an application that will transform the Android device into a baby toy. The toy will display baby-friendly images in a 30 second loop on the center of the screen. The application will also have 4 buttons placed in the 4 corners on the screen. These buttons will act as a baby proof lock: the application exits only when the user presses the 4 buttons in sequence, with no longer than a 1 second gap between two button pushes. Pressing any of the buttons should not interfere with the images that will show on the screen to entertain the baby.



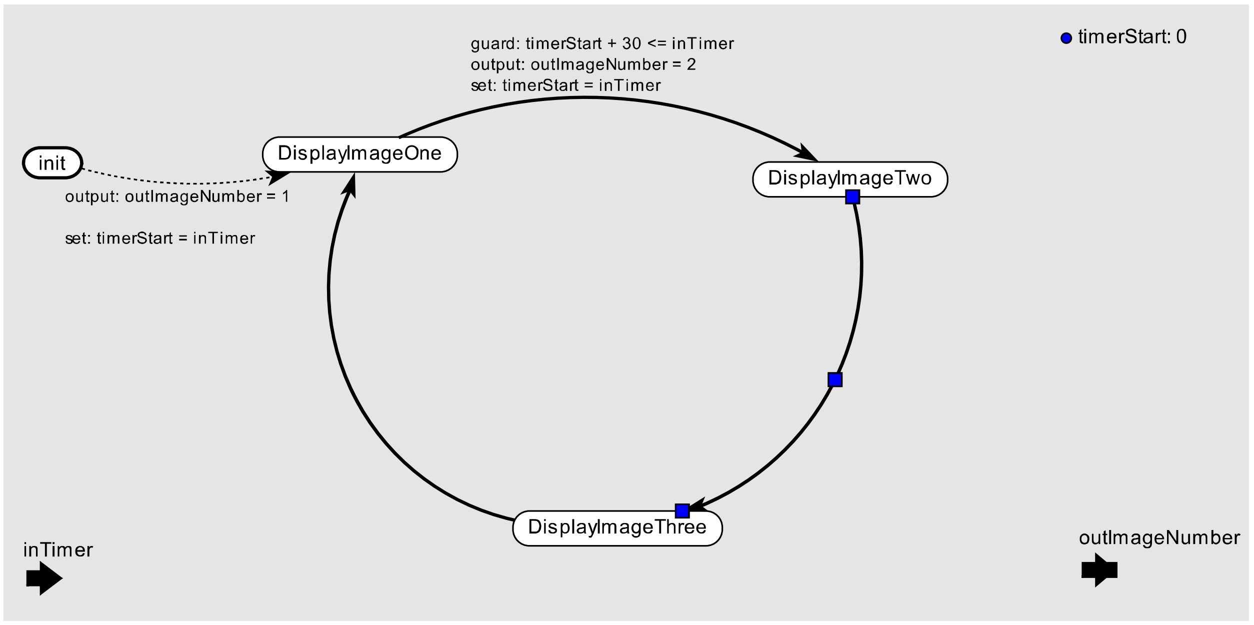
4.1.1 Starter Model

The starter model for the baby toy application contains four buttons, two output boxes, one timer, and two (empty) FSMs. **It's your responsibility to implement the content of these FSMs, one of which corresponds to the image rotation functionality and the other to the detection of the exit.**



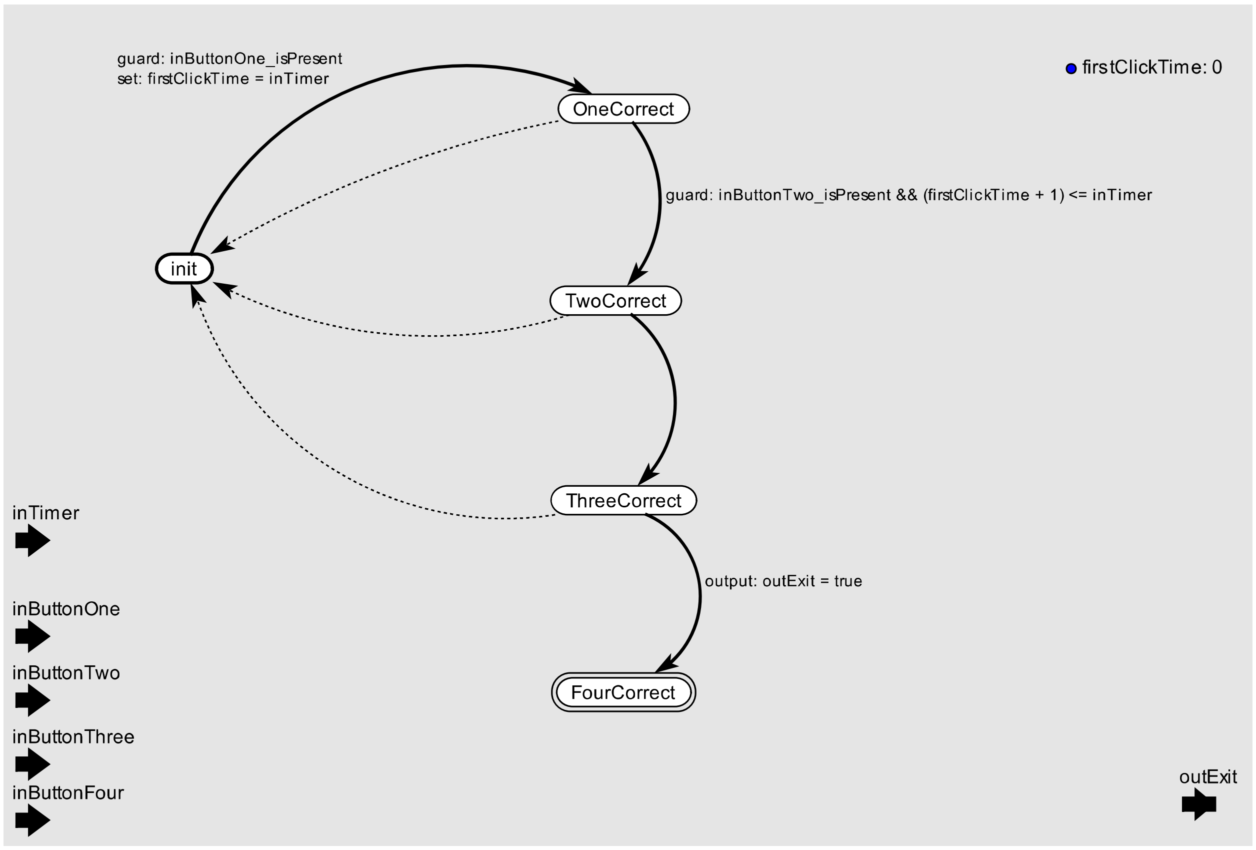
4.1.2 Modeling

As in the security access app, we want to design and model the baby toy before we begin any implementation. We rely on Ptolemy II to produce a FSM, and the associated machinery necessary to execute and test it. For this application, we need to also model the periodic image rotations and to model the time constraint between consequtive button pushes. Both of these require to model the concept of time in our FSMs. To model the passage of time more effectively we use FSMs with additional capabilities, called extended FSM, which can set and get variables.


guard: timerStart + 30 <= inTimer
output: outImageNumber = 2
set: timerStart = inTimer
timerStart: 0
init
DisplayImageOne
DisplayImageTwo
DisplayImageThree
output: outImageNumber = 1
set: timerStart = inTimer
inTimer
outImageNumber

Above is an incomplete version of the image rotation part of the Baby Toy model. Read the passage below and study the model carefully, with the help of your instructor, and complete the missing guards/outputs.

The above model encodes the image rotations, which occur every 30 seconds. The above finite state machine has one input port, which is a discrete counter/timer, and one output port that denotes the image that will appear on the Android device's screen. The counter input begins at some number, for instance 0, and increses by one. We record the initial count of the timer into a variable called `timerStart`. The FSM transitions only when it has measured a change in the timer of at least 30 counts (corresponding to 30 seconds), starting from `timerStart`. Each transition reinitializes the `timerStart` variable to begin the countdown, and outputs an number corresponding to a different image that will appear of the screen. Also notice that this FSM does not have a final state, as in this application the image rotation occures infinitely. Exiting the application is goverened by the 4 buttons in the corners of the screen, which are encoded in the following FSM.



Same as before, above is an incomplete version of the button pressing part of the Baby Toy model. Read the passage below and study the model carefully, with the help of your instructor, and complete the missing guards/outputs.

This FSM has five inputs, four of which are for the button pushes, and one that encodes a discrete timer/counter. The timer's purpose is to measure out the 1 second interval necessary between the button pushes. The FSM proceeds from state to state on a correct button push much like the previous security device FSM. The only difference is that we aren't worring about detecting wrong sequences as much as before. Notice that pushing a wrong button doesn't take the FSM into a "wrong" state. Rather, failing to press the next button within the necessary 1 seconds interval returns the FSM back to the initial state, which corresponds to no pushed buttons.

In modeling this application, we supplied separate FSM models encoding the four button sequence needed to end the application and the 30s image rotation. Note that two models for one application can be a bad idea for applications where models share data or state. In this application, the two FSM are completely independent and do not share any data, so constructing separate models is reasonable.

4.1.3 Implementation

In the Security Access Device app, you learned how to create buttons and labels and update their properties (e.g. *Id, Text, etc.*). This application uses the same principles, with the addition of a new widget, called a `ImageView` that is used to display the image in the middle of the screen. This widget has a `setView` method/property that can be used to select an image file to display.

Image files in a few common formats (e.g. jpeg, png) need to be added into your project's *res* directory. Here a number of *drawable* subdirectories indicate the resolution of the image (e.g. *hdpi - high dots per inch*). Usually, most images you will come accross can safely be placed in the *mdpi - medium dpi* subdirectory.

It's your task to retrieve three baby toy images from the Internet. In doing so, please be aware of copyright issues - i.e. don't just take a picture without the author's permission. A good site with freely available pictures is Wikimedia

Now that we are finished with the user interface, we can direct our attention to implementing the logic of the application. Again, we want to make heave use of the finite state machines we modeled in Ptolemy.

Stop! Before you continue make sure you understand the concept of threads and how they are implemented in Java

The image rotation is actually fairly straightforward to implement, as the timing aspects of it are managed by an Android handler object – `android.os.Handler`. These objects manage execution threads by scheduling them to begin executing in the activity's `onCreate` method, and, once started, repeatedly re-schedule the thread to execute at a particular time offset. In our example implementation given below, we use a 1 second scheduling interval for both the `rotateImageTask` and `detectExitTask` threads. Each of these threads implement one of the FSMs we modeled before. The `rotateImageTask` is quite simple, as it measures the number of executions of this thread in the `inTimer` variable. Once the count reaches 30, this corresponds to 30 seconds elapsed, and we rotate the image.

```
enum ImageState {
    DisplayImageOne,
    DisplayImageTwo,
    DisplayImageThree
}

private ImageState imageStatus = ImageTypes.DisplayImageOne;
private ImageView image = null;
private Handler threadHandler = null;

private Runnable rotateImageTask = new Runnable() {

    private int inTimer = 0;
    private int timerStart = 0;

    public void run() {
        long millisElapsed = SystemClock.uptimeMillis();
        inTimer++;
        image = (ImageView) findViewById(R.id.ImageViewId);
        if((timerStart + 30) <= inTimer) {
            timerStart = inTimer;
            if(imageStatus == ImageState.DisplayImageOne) {
                imageStatus = ImageState.DisplayImageTwo;
                image.setImageResource(R.drawable.filename1);
            }
            else if(imageStatus == ImageState.DisplayImageTwo) {
                imageStatus = ImageState.DisplayImageThree;
                image.setImageResource(R.drawable.filename2);
            }
            else if(imageStatus == ImageState.DisplayImageThree) {
                imageStatus = ImageState.DisplayImageOne;
                image.setImageResource(R.drawable.filename3);
            }
        }

        threadHandler.postAtTime(this, millisElapsed + 1000); //post every 1 second
    }
};
```

Now, we are left with implementing the application exit via ordered presses of the four buttons in the corners, with no more than one seconds of time between two presses. We have some experience dealing with buttons from the security access device in our first Android app. To this, we need to add the ability to measure out a 1 second time interval between button pushes. As before, we use an enumeration type defining each of the states.

```
enum AppState {
    Init,
    OneCorrect,
    TwoCorrect,
    ThreeCorrect
}

private AppState state = AppState.Init;

private boolean b1=false;
private boolean b2=false;
private boolean b3=false;
private boolean b4=false;

//Button check thread
private Runnable detectExitTask = new Runnable() {
    public void run() {
        long millisElapsed = SystemClock.uptimeMillis();

        if(b1 && state == AppState.Init) {
            state = AppState.OneCorrect;
            b1 = false;
        }
        else if(b2 && state == AppState.OneCorrect) {
            state = AppState.TwoCorrect;
            b2 = false;
        }
        else if(b3 && state == AppState.TwoCorrect) {
            state = AppState.ThreeCorrect;
            b3 = false;
        }
        else if(b4 && state == AppState.ThreeCorrect) {
            //TODO: use activity.finish() here
        }
        else {
            state = AppState.Init;
            b1 = b2 = b3 = b4 = false;
        }
        threadHandler.postAtTime(this, millisElapsed + 1000); //post every 1 second
    }
};
```

The above code implements the FSM from our model by using the `exitState` variable to denote the current state. Four boolean variables, `b1` through `b4` are set to true whenever a specific button is clicked. This is performed in each of the button's `onClickListener`, which was registered in each of the button's `onClick` property in the user interface, and is shown below. The handler thread is scheduled to execute every one second, and when it does so it either progresses through the clicked buttons of the FSM (e.g. `OneCorrect` to `TwoCorrect`) or reverts back to the `Init` state. If it returns to the `Init` state, it will reset all of the button clicks by setting `b1` to `b4` to false. This has the effect that if the application did not receive the correct button click in the one second interval, it returns back to the beginning and erases all of the previous clicks from the state machine.

```
/** Called when the activity is first created. */
public void onCreate(Bundle savedInstanceState) {
    ...
    //some android stuff is here, don't change it
    ...

    //start the threads
    threadHandler = new Handler();
    threadHandler.removeCallbacks(rotateImageTask);
    threadHandler.postDelayed(rotateImageTask, 100);
    threadHandler.removeCallbacks(detectExitTask);
    threadHandler.postDelayed(detectExitTask, 100);
}

public void clickedButtonOne(View view) {
  b1 = true;
}

public void clickedButtonTwo(View view) {
  b2 = true;
}

public void clickedButtonThree(View view) {
  b3 = true;
}

public void clickedButtonFour(View view) {
  b4 = true;
}
```

4.1.4 Application Extensions

These are suggestions for possible enhancements to the base baby rattle application. Students are expected to improve the model as well as produce a working implementation of the following extension.

- **Exit the application properly.** Figure out how to use the activity.finish method to exit the application instead of going in a nonresponsive state. You will need to read a bit of Android documentation (google it!) and figure out how to get an handle of the activity object to where you need it in your code.
- **Baby rattle.** Babies like things that rattle, so make the phone rattle at 30 second intervals that alternate with the image changes. In other words, every 15 seconds the app will either change the image or rattle.