Atlas Demonstration Toolbox

This document describes the example Atlas use cases provided in the Demonstration Toolbox. Atlas is a fully general software analysis and visualization platform. Users write Atlas queries to produce interesting graphs and solve specific use cases. The use cases described here are the tip of the iceberg of what Atlas can do, and are meant to provide a glimpse of what is possible.

Prerequisites

EnSoft Atlas

The Atlas Demonstration Toolbox requires an Eclipse install with the EnSoft Atlas plugin. If you do not have a copy of Atlas, please visit EnSoft's request page.

Atlas Request: http://www.ensoftcorp.com/atlas_request/

Android SDK

The Atlas Demonstration Toolbox provides Android apps as analysis examples. For the best experience, please download the Android SDK and install the ADT plugin for your Eclipse install.

Android SDK: http://developer.android.com/sdk/index.html#download

ADT Plugin: http://developer.android.com/sdk/installing/installing-adt.html

Demonstration Toolbox

To use the Demonstration Toolbox, you will need a copy of the toolbox project. Use the link below, download the project, and import it into your Eclipse workspace.

Demonstration Toolbox: https://github.com/EnSoftCorp/Demonstration-Toolbox

Code Comprehension Demo

Scripts: src/com.ensoftcorp.atlas.java.demo.comprehension.ComprehensionUtils

Example App: app/connectbot

Setup:

1. Import ConnectBot, index with Atlas, and restart the Interpreter view.
2. Open ComprehensionUtils.scala for viewing.

Software today is large and complex. In order to maintain or even understand a large code base, software engineers need effective comprehension tools to answer questions such as:

What methods call method M?

Where does type T sit in the inheritance hierarchy?

What data may flow to variable V?

Some "canned" tools exist for answering questions like this. Unfortunately, these tools tend to be hard-coded for a set of very specific use cases, and cannot be customized or used for other purposes. With Atlas, we can write customized scripts to answer these questions, without diminishing the flexibility or generality of the tool!

Take a look at ComprehensionUtils, a set of Atlas scripts written in a few minutes to answer comprehension questions. We see several public methods defined: callGraph, dataFlow, interactions, overrides, and typeHierarchy. Most of these call a private method called bidirectional, which takes a given origin point in the software graph and a variable number of edge kinds.

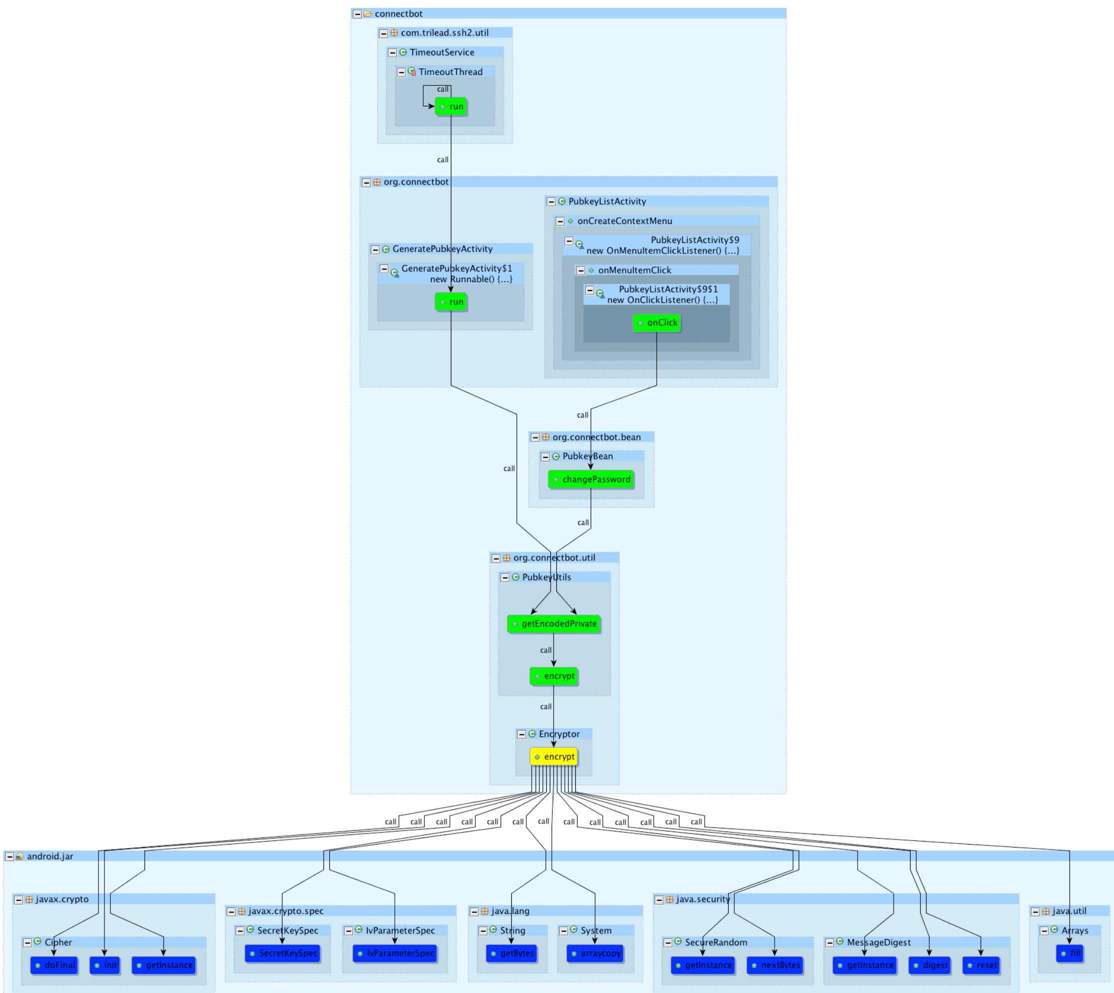
```
def callGraph(start:Q):DisplayItem = {
     bidirectional(start, Edge.CALL)
}

private def bidirectional(start:Q, edgeKinds:String*):DisplayItem = {
     var context = universe.edgesTaggedWithAny(edgeKinds:_*)
     context = differenceEdges(context,context.edgesTaggedWithAny(
     Edge.PER_CONTROL_FLOW))

     var ancestors = context.reverse(start)
     var decendents = context.forward(start)

     var artist = new Artist
     artist.addColor(start, Color.YELLOW, PaintMode.NODES)
     artist.addColor(ancestors difference start, Color.GREEN, PaintMode.NODES)
     artist.addColor(decendents difference start, Color.BLUE, PaintMode.NODES)
     new DisplayItem(ancestors union decendents, artist.getHighlighter)
}
```

Bidirectional selects all edges of the given kinds, performs a forward and reverse traversal from the given origin, and returns the two traversals together with some coloring.

Let's give this a try. Open the class org.connectbot.util.Encryptor, and select the encrypt method. Suppose we're a developer and we just made a change to this method. Now we'd like to know the impact of our change by looking at a call graph of the method. On the Interpreter, enter the following to invoke our callGraph script:

```
callGraph(selected).show
```

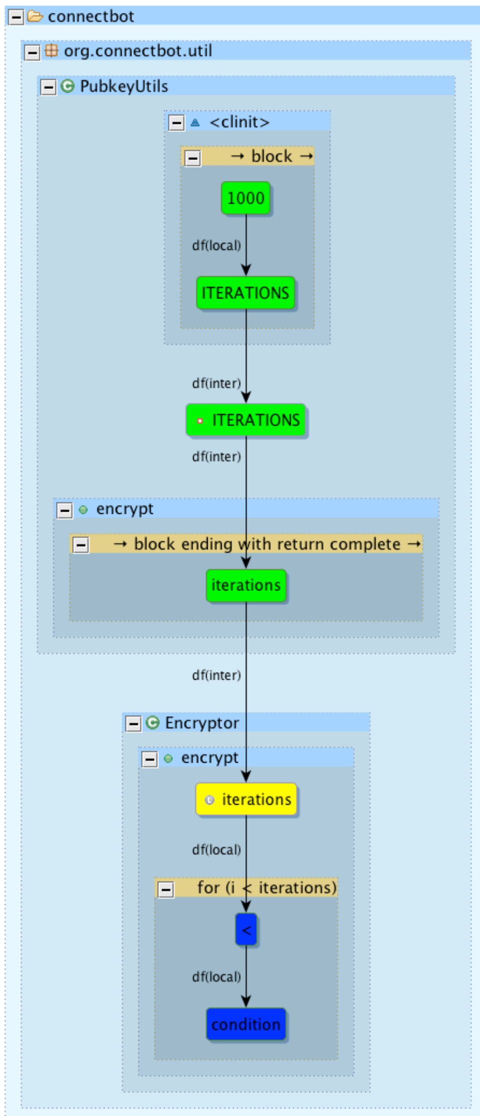
This query produces and displays the following graph for us.



Notice that the encrypt method itself is highlighted in yellow, its reverse call graph in green, and its forward call graph in blue. Now, suppose we made a specific change to the way encrypt uses its parameter, iterations. Perhaps we would like to see a data flow graph of where this data comes from and flows to. Returning to the code for encrypt, select the iterations parameter and entered the following on the interpreter:

```
dataFlow(selected).show
```

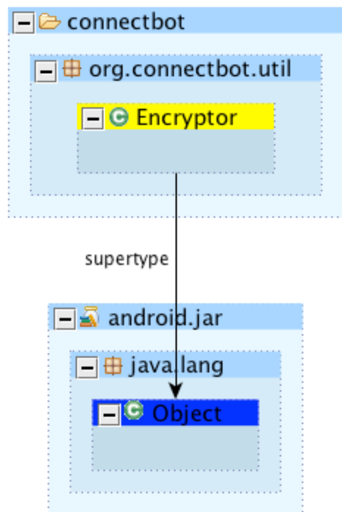
This query produces and displays the following graph:



Again, the parameter itself is highlighted in yellow, while the forward and reverse data flow graphs are blue and green, respectively. Finally, suppose we want to see the type hierarchy for Encryptor to ensure that we've made our changes in the correct spot. Select the Encryptor class (either in the previous graph, or else in its source code), and enter the following on the interpreter:

```
typeHierarchy(selected).show
```

This query produces and displays the following graph:



We see that Encryptor directly inherits from Object, and nothing extends it. Now that we have used Atlas to evaluate the potential impact of our changes, we are comfortable that all is well.

Notice that these features are **not** part of Atlas... we added them ourselves when we wrote the scripts in ComprehensionUtils! This means that we have the power to add more features, tweak existing functionality, remove functionality, etc. to our heart's content! Other code comprehension tools do not offer customization like this.

Safe Synchronization Demo

Scripts: src/com.ensoftcorp.atlas.java.demo.synchronization.RaceCheck

Example App: app/Synchronization-Example

Setup:

1. Import Synchronization-Example, index with Atlas, and restart the Interpreter view.
2. Open RaceCheck.scala for viewing.
3. Open the only class in the example, ProducerConsumer, for viewing.

In the previous example, we showed how Atlas can be used to perform fairly vanilla code comprehension tasks. Let's do something a bit more customized. Suppose we're the authors of a multithread application which employs the producer/consumer design pattern. We know that any shared data structures modified by multiple threads should be locked for exclusive access. Have we done this correctly in our application? We can write an Atlas script to check!

```
public void dangerous() {
     sharedBuffer.add(someItem);
}
public void safe() {
     synchronized(sharedBuffer){
          sharedBuffer.add(someItem);
     }
}
```

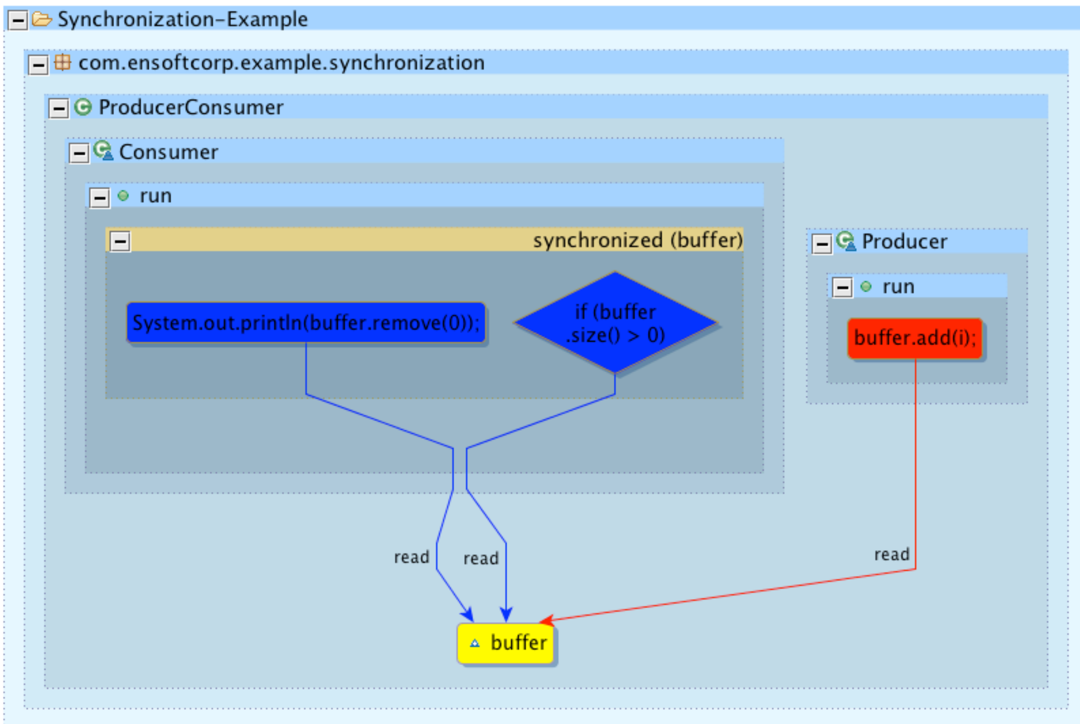
Consider RaceCheck.scala, inside which is a method called raceCheck. This method takes an expression representing some shared data structures whose access we should check. Then, it finds various control flow blocks from which reads and writes to that structure are made. For each of these accesses, raceCheck determines whether it is made under a synchronized block or not. Those accesses which happen under synchronized blocks are “safe” and are colored blue, while potentially dangerous accesses are colored red.

```
def raceCheck(sharedTokens:Q):DisplayItem = {
     var decContext = edges(Edge.DECLARES)
     var rwContext = edges(Edge.READ, Edge.WRITE)
     rwContext = differenceEdges(rwContext,
          rwContext.edgesTaggedWithAny(Edge.PER_METHOD))

     var accessors = stepTo(rwContext, sharedTokens) union
          stepFrom(rwContext, sharedTokens)
     accessors = accessors difference declarations(methods("<init>")
          union methods("<clinit>"))
     var synchronizeBlocks =
          universe.nodesTaggedWithAny(Node.SYNCHRONIZED)
     var decBySynchronized = decContext.forwardStep(synchronizeBlocks)
     accessors = accessors difference
          (edges(Edge.CONTROL_FLOW).reverseStep(decBySynchronized)
          difference decBySynchronized)

     var goodAccesses = empty
     var badAccesses = empty
     var ge = 0
     for(ge <- accessors.eval.nodes){
          var geQ = toQ(toGraph(ge))
          var revDec = decContext.reverse(geQ)
          if((revDec intersection synchronizeBlocks).eval.nodes.size
                > 0){
               goodAccesses = goodAccesses union geQ
          } else{
               badAccesses = badAccesses union geQ
          }
     }
     …
}
```

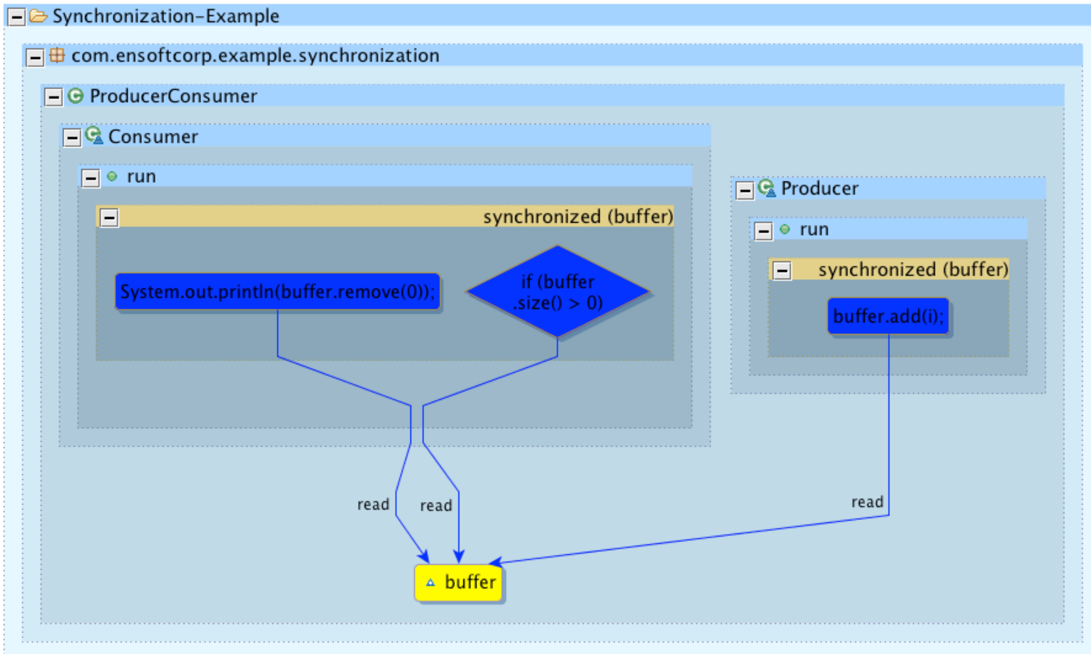
Let’s use the raceCheck Atlas script to check the thread safety of our shared buffer data structure. Select the buffer field, then enter the following query in the Interpreter:

```
raceCheck(selected).show
```

This query produces and displays the following graph:



It looks like our script has detected an error! Click on the red control flow block to take us to the error location in the code. We see that someone has commented out the synchronized block in the Producer thread. Re-add the block, save the file, re-index the project, and restart your interpreter. Now, let's see if our script shows that the problem is fixed. Repeat your last query.



Looks good! Our script has confirmed our fix. As with the previous demonstration, notice that this functionality was **not** provided by Atlas. We created a tool with this functionality by writing a custom script! Atlas allows us to add new features and tools on-the-fly, unlike other analysis tools.

API Compliance Demo

Scripts: src/com.ensoftcorp.atlas.java.demo.apicompliance.APICompliance

Example App: app/Android-Audio-Capture-Example

Setup:

1. Import Android-Audio-Capture-Example, index with Atlas, and restart the Interpreter view.
2. Open APICompliance.scala for viewing.
3. Open the only class in the example, MainActivity, for viewing.

Here we will show another highly-customized analysis example using Atlas. Suppose we are the authors of a brilliant new Android app which records audio. Android provides a set of APIs for audio capture (http://developer.android.com/guide/topics/media/audio-capture.html). A sequence of nine method calls, setting up various aspects of the recording, is required. Suppose we wish to know if our Android app uses these APIs correctly. We can write an Atlas script to check!

```
"
Audio capture from the device is a bit more complicated than audio and video
playback, but still fairly simple:

1. Create a new instance of android.media.MediaRecorder.
2. Set the audio source using MediaRecorder.setAudioSource(). You will probably
want to use MediaRecorder.AudioSource.MIC.
3. Set output file format using MediaRecorder.setOutputFormat().
4. Set output file name using MediaRecorder.setOutputFile().
5. Set the audio encoder using MediaRecorder.setAudioEncoder().
6. Call MediaRecorder.prepare() on the MediaRecorder instance.
7. To start audio capture, call MediaRecorder.start().
8. To stop audio capture, call MediaRecorder.stop().
9. When you are done with the MediaRecorder instance, call
MediaRecorder.release() on it. Calling MediaRecorder.release() is always
recommended to free the resource immediately.
"
```

Take a look at APICompliance.scala, which contains a custom method called androidAudioCapture. This method selects the nine relevant API methods, then defines an expected ordering for calls. Next, it builds a control flow graph of our application, and identifies where the API call events occur. Finally, each call event is sorted into a “good” or “bad” group based upon whether all prerequisite calls were made prior to making this call. In the resulting graph, correct API calls are shown in blue, while incorrect calls are shown in red.

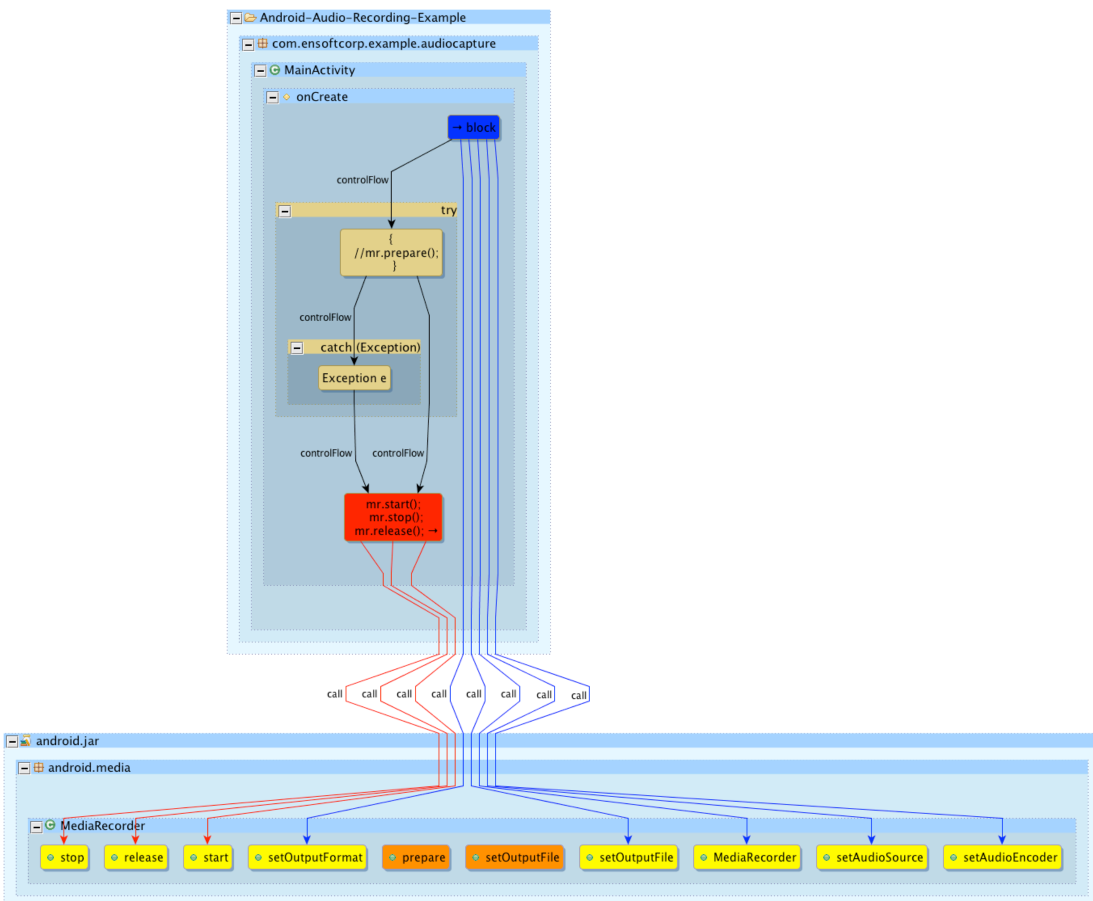
```
def androidAudioCapture():DisplayItem = {
     var callContext = edges(Edge.CALL)
     var cfContext = edges(Edge.CONTROL_FLOW)

     var mediaRecorder =
          methodSelect("android.media","MediaRecorder","MediaRecorder")
     var setAudioSource =
          methodSelect("android.media","MediaRecorder","setAudioSource")
     var setOutputFormat =
          methodSelect("android.media","MediaRecorder","setOutputFormat")
     var setOutputFile =
          methodSelect("android.media","MediaRecorder","setOutputFile")
     var setAudioEncoder =
          methodSelect("android.media","MediaRecorder","setAudioEncoder")
     var prepare = methodSelect("android.media","MediaRecorder","prepare")
     var start = methodSelect("android.media","MediaRecorder","start")
     var stop = methodSelect("android.media","MediaRecorder","stop")
     var release = methodSelect("android.media","MediaRecorder","release")
     var apis = mediaRecorder.union(setAudioSource, setOutputFormat,
          setOutputFile, setAudioEncoder, prepare, start, stop, release)

     val requisiteOrder = Array(mediaRecorder, setAudioSource,
          setOutputFormat, setOutputFile, setAudioEncoder, prepare, start,
          stop, release)

     var cfGraph =
          inducedGraph(app.nodesTaggedWithAny(Node.CONTROL_FLOW), cfContext)
```

```
     var apiCalls = callContext.forwardStep(cfGraph) intersection
          callContext.reverseStep(apis)

     cfGraph = cfGraph difference (cfGraph difference
          cfGraph.between(apiCalls, apiCalls))

     var cfEnhanced = cfGraph union apiCalls

     val (goodCalls, badCalls) = precedenceCheck(requisiteOrder, cfEnhanced)
     …
}
```

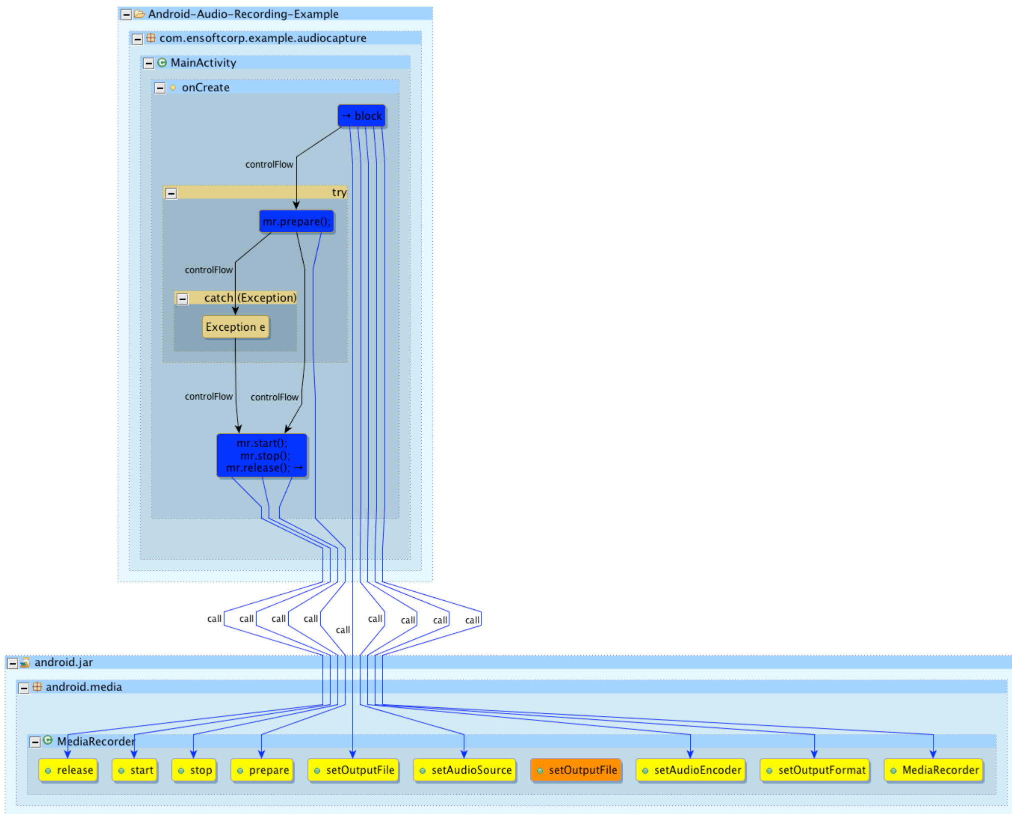
Let's run this on our Android app to see how we've done. On the Interpreter, run:

`androidAudioCapture.show`

This query produces and displays the following graph:



Our Atlas script detected problems with our app. Click on the red control flow block from which the errant calls are made. We see that we have forgotten to call prepare prior to starting the recording. Uncomment the prepare call, save the file, re-index the project, and restart your interpreter. Now, run the same query again.



Looks good! Our Atlas script has confirmed our fix, and all API calls are now made in the correct order. As with the previous demonstrations, notice that this functionality was **not** provided by Atlas. We created a tool with this functionality by writing a custom script! Atlas allows us to add new features and tools on-the-fly, unlike other analysis tools.

Resources

Android SDK: http://developer.android.com/sdk/index.html#download
ADT Plugin: http://developer.android.com/sdk/installing/installing-adt.html
Atlas Product Page: http://www.ensoftcorp.com/atlas/
Atlas Request: http://www.ensoftcorp.com/atlas_request/
Demonstration Toolbox: https://github.com/EnSoftCorp/Demonstration-Toolbox
Email EnSoft: contactus@ensoftcorp.com