Daikon Tutorial

December 9, 2016

1 Introduction

Daikon is an implementation of dynamic detection of likely invariants; that is, the Daikon invariant detector reports likely program invariants. An invariant is a property that holds at a certain point or points in a program; these are often seen in assert statements, documentation, and formal specifications. Invariants can be useful in program understanding and a host of other applications. Examples include

- ‘`x.field > abs(y)`’
- ‘`y = 2*x+3`’
- ‘`array a is sorted`’
- ‘`for all list objects lst, lst.next.prev = lst`’
- ‘`for all treenode objects n, n.left.value < n.right.value`’
- ‘`p != null => p.content in myArray`’

Dynamic invariant detection runs a program, observes the values that the program computes, and then reports properties that were true over the observed executions. Daikon is freely available for download from download-site. The distribution includes both source code and documentation, and Daikon’s license permits unrestricted use. Many researchers and practitioners have used Daikon; those uses, and Daikon itself, are described in various publications.

2 Installing Daikon

If you have run `integration-test2/fetch.py` (or the underlying `integration-test2/fetch_dependencies.sh`) then your Daikon installation should be ready to go. If you wish to install Daikon for further exploration on your own, see the Installing-Daikon section of the Daikon User Manual.

3 Overview of running Daikon

Detecting invariants involves two steps:

1. Obtain one or more data trace files by running your program under the control of a front end (also known as an instrumenter or tracer) that records information about variable values. You can run your program over one or more inputs of your own choosing, such as regression tests or a typical user input session. You may choose to obtain trace data for only part of your program; this can avoid inundating you with output, and can also improve performance.
2. Run the Daikon invariant detector over the data trace files. This detects invariants in the recorded information. You can view the invariants textually, or process them with a variety of tools.

In order to detect invariants in a Java program, first run the program using the DynComp front end, then pass the resulting `.decls` file to Chicory. Finally, run Daikon itself to detect invariants. With the `--daikon` option to Chicory, a single command performs the last two steps.

For example, if you usually run

```
java mypackage.MyClass arg1 arg2 arg3
```

then instead you would run

```
java daikon.DynComp mypackage.MyClass arg1 arg2 arg3
java daikon.Chicory --daikon \
                    --comparability-file=MyClass.decls-DynComp \
                    mypackage.MyClass arg1 arg2 arg3
```

and the Daikon output is written to the terminal.

4 Running Daikon on a simple program

The Daikon distribution contains some sample programs that will help you get practice in running Daikon.

To detect invariants in the `StackAr` sample program, perform the following steps after installing Daikon.

1. Compile the program with the `-g` switch to enable debugging symbols.

   ```
   cd $DAIKONDIR/examples/java-examples/StackAr
   javac -g DataStructures/*.java
   ```

2. Run the program under the control of DynComp to generate comparability information in the file `StackArTester.decls-DynComp`.

   ```
   java -cp .:$CLASSPATH daikon.DynComp --no-cset-file
   DataStructures.StackArTester
   ```

3. Run the program a second time, under the control of the Chicory front end. Chicory observes the variable values and passes them to Daikon. Daikon infers invariants, prints them, and writes a binary representation of them to file `StackArTester.inv.gz`.

   ```
   java -cp .:$CLASSPATH daikon.Chicory --daikon \
         --comparability-file=StackArTester.decls-DynComp \
         DataStructures.StackArTester
   ```

If you wish to have more control over the invariant detection process, you can split the third step above into multiple steps. Then, step 3 would become:

3. Run the program under the control of the Chicory front end, including comparability information, in order to create a trace file named `StackArTester.dtrace.gz`.

   ```
   java -cp .:$CLASSPATH daikon.Chicory \
         --comparability-file=StackArTester.decls-DynComp \
         DataStructures.StackArTester
   ```

4. Run Daikon on the trace file.

   ```
   java daikon.Daikon StackArTester.dtrace.gz
   ```

   (Note the classpath (`-cp`) argument is not needed as we are not running the `StackArTester` program.)

   You could capture a text copy of the invariants with:

   ```
   java daikon.PrintInvariants StackArTester.inv.gz > inv.log
   ```

Daikon provides many options for controlling how invariants are printed. Often, you may want to print the same set of invariants several different ways. However, you only want to run Daikon once, since it may be very time-consuming. The `PrintInvariants` utility prints a set of invariants from a `.inv` file.

`PrintInvariants` is invoked as follows:

```
java daikon.PrintInvariants [flags] inv-file
```

See the Printing-invariant section of the Daikon User Manual for details about using this tool.

5 Understanding the invariants

This section examines some of the invariants for the `StackAr` example. This program is an array-based stack implementation. Take a look at `DataStructures/StackAr.java` to get a sense of the implementation. Now, look at the sixth section of Daikon output.

```
===========================================================================
StackAr:::OBJECT
this.theArray != null
this.theArray.getClass().getName() == java.lang.Object[].class
this.topOfStack >= -1
this.topOfStack <= size(this.theArray[])-1
===========================================================================
```

These four annotations describe the representation invariant. The array is never null, and its runtime type is `Object[]`. The `topOfStack` index is at least -1 and is less than the length of the array.

Next, look at the invariants for the `top()` method. `top()` has two different exit points, at lines 74 and 75 in the original source. There is a set of invariants for each exit point, as well as a set of invariants that hold for all exit points. Look at the invariants when `top()` returns at line 75.

```
===========================================================================
StackAr.top():::EXIT75
return == this.theArray[this.topOfStack]
return == this.theArray[orig(this.topOfStack)]
return == orig(this.theArray[post(this.topOfStack)])
return == orig(this.theArray[this.topOfStack])
this.topOfStack >= 0
return != null
===========================================================================
```

The return value is never null, and is equal to the array element at index `topOfStack`. The top of the stack is at least 0.

6 DynComp dynamic comparability (abstract type) analysis for Java

While Daikon can be run using only the Chicory front end, it is highly recommend that DynComp be run prior to Chicory. The DynComp dynamic comparability analysis tool performs dynamic type inference to group variables at each program point into comparability sets (see Section "Program point declarations" in *Daikon Developer Manual* for the numeric representation format of these sets.) All variables in each comparability set belong to the same "abstract type" of data that the programmer likely intended to represent, which is a richer set of types than the few basic declared types (e.g., int, float) provided by the language.

Without comparability information, Daikon attempts to find invariants over all pairs (and sometimes triples) of variables present at every program point. This can lead to two negative consequences: First, it may take lots of time and memory to infer all of these invariants, especially when there are many global or derived variables present. Second, many of those invariants are true but meaningless because they relate variables which conceptually represent different types (e.g., an invariant such as `winterDays < year` is true but meaningless because days and years are not comparable).

It should be noted that the performance of Daikon with and without the use of DynComp can be significant. A 10-30X improvement in the running time of Daikon is typical.

6.1 Understanding DynComp

To get a sense of how DynComp helps eliminate uninteresting output, take a look at the invariants for the entry point of the `createItem(int)` method. (This first example is what you would have gotten without running DynComp.)

```
===========================================================================
DataStructures.StackArTester.createItem(int):::ENTER
phase >= 0
DataStructures.StackArTester.s.topOfStack < size(DataStructures.StackArTester.s.theArray[])-1
phase <= size(DataStructures.StackArTester.s.theArray[])
phase != size(DataStructures.StackArTester.s.theArray[])-1
===========================================================================
```

The value of `phase` is always less than the size of `theArray[]`. While this is true for the observed executions, it is not a helpful invariant, since `phase` and `size(theArray[])` represent different abstract types. Although they are both `int`s, comparing the two is not meaningful, so this invariant, among others, is omitted from the output when Daikon is run with DynComp.

```
===========================================================================
DataStructures.StackArTester.createItem(int):::ENTER
phase >= 0
DataStructures.StackArTester.s.topOfStack < size(DataStructures.StackArTester.s.theArray[])-1
===========================================================================
```

7 Running Daikon on programs from the corpus

The PASCALI corpus is a subset of the Leidos corpus. To run Daikon on a program from the corpus, using a test suite automatically generated by Randoop, run the `run_dyntrace.py` python script. For example, to run Daikon on the `react` project:

```
./run_dyntrace.py react
```

You may wish to redirect the output to a log file, or use the `PrintInvariants` tool to get a copy of the invariants:

```
java daikon.PrintInvariants corpus/react/dljc-out/test-classes1/invariants.gz > inv.txt
```

The next section shows some of the invariants that Daikon outputs for these programs.

7.1 imagej invariant examples

```
===========================================================================
ij.IJ:::CLASS
ij.IJ.df[].getClass().getName() elements == java.text.DecimalFormat.class
size(ij.IJ.df[]) == 10
===========================================================================
```

These invariants show that the '`df[]`' array always has 10 elements, and furthermore each of those elements is an object of type '`DecimalFormat`'.

7.2 jreactphysics3d Invariant examples

```
===========================================================================
net.smert.jreactphysics3d.constraint.FixedJointInfo:::OBJECT
this.type == net.smert.jreactphysics3d.constraint.JointType.FIXEDJOINT
this.positionCorrectionTechnique ==
net.smert.jreactphysics3d.configuration.JointsPositionCorrectionTechnique.NON_LINEAR_GAUSS_SEIDEL
===========================================================================
```

The ‘`FixedJointInfo`’ subclass always has type ‘`FIXEDJOINT`’, which verifies an expected property. In addition, the test suite does only uses ‘`NON_LINEAR_GAUSS_SEIDEL`’, without testing other possibilites for the position correction technique.

```
===========================================================================
net.smert.jreactphysics3d.engine.CollisionWorld:::OBJECT
this == this.collisionDetection.world
===========================================================================
```

This invariant shows that the ‘`CollisionWorld`’ and ‘`CollisionDetection`’ objects refer to one another — and they do so correctly! Thus, ‘`this`’ is always identical to ‘`this.collisionDetection.world`’. This shows how multiple expressions can be used to access the same data.

7.3 react Invariant examples

```
===========================================================================
com.flowpowered.react.collision.shape.BoxShape:::OBJECT
this.mType ==
com.flowpowered.react.collision.shape.CollisionShape$CollisionShapeType.BOX
this.mExtent.x one of { 0.96, 628.96 }
this.mExtent.y one of { 9.96, 2812.96 }
this.mExtent.z one of { 9.96, 51.96 }
this.mNbSimilarCreatedShapes one of { -1, 0 }
size(com.flowpowered.react.collision.shape.CollisionShape$CollisionShapeType.$VALUES[]) == 6
this.mExtent.x < this.mExtent.y
this.mExtent.x != this.mExtent.z
this.mExtent.x > this.mMargin
this.mExtent.y >= this.mExtent.z
this.mExtent.y > this.mMargin
this.mExtent.z > this.mMargin
===========================================================================
```

7.4 thumbnailinator Invariant examples

```
===========================================================================
net.coobird.thumbnailator.makers.FixedSizeThumbnailMaker:::OBJECT
this.width >= 0
this.height >= 0
this.imageType one of { -1, 2 }
this.imageType != 0
===========================================================================
net.coobird.thumbnailator.makers.ThumbnailMaker$ReadinessTracker.isSet(java.lang.String):::EXIT
(return == true)  ==>  (arg0.toString one of { "keepRatio", "scale", "size" })
===========================================================================
net.coobird.thumbnailator.makers.ThumbnailMaker.defaultImageType():::EXIT
this.imageType == 2
return.imageType == 2
this.imageType >= orig(this.imageType)
===========================================================================
```