CSE 331 Software Design and Implementation

Handout T7: *Daikon invariant detector*

Contents:

Overview

The Daikon invariant detector reports properties that hold at procedure entries and exits. The reported properties are similar to those in specifications or assert statements: representation invariants, preconditions ("requires" clauses), or postconditions ("effects" clause). Examples include "`x == abs(y)`", "`i < myArray.length`", or "`myArray is sorted`". These invariants can be useful in program understanding, testing, and a host of other applications.

Daikon runs your program, examines the values your program computes, and uses machine learning to find patterns and relationships among those values. Daikon reports properties that were true for all the executions that it observed.

Daikon's output differs from the specifications that you write in two important ways:

- Daikon's output may contain properties that the true specification does not. (In other words, Daikon's output may be stronger (more specific) than the true specification.)

  Daikon reports what it observed during execution, but properties may be true of a given execution (especially of a poor test suite) that are not true in general. For example, Daikon may report that all arguments to a procedure are positive. Maybe they truly must be positive, or maybe your test suite is poor, and it should really pass negative values to the procedure.

- Daikon's output may be missing some properties that appear in the true specification. (In other words, Daikon's output may be weaker (less specific) than the true specification.)

  Daikon can only report a limited number of properties; its learning algorithm is not as powerful or clever as the human mind.

Despite these limitations, Daikon's output is often a remarkably close approximation of the specification, and examining Daikon's output can assist you in understanding problems with your test suites, your specifications, or your code.

You can obtain more information about Daikon from its website. In particular, you can read its manual (HTML, PDF). If you have any problems with Daikon, please send mail to daikon-developers@pag.csail.mit.edu, post to the forums, or ask a CSE 331 staff member.

Running Daikon

We provide an Ant target for running Daikon. It will work for any assignment:

```
cd hw1
ant daikon
```

(You should not run `ant daikon` until after `ant test` executes with no test failures.)

`ant daikon` produces a representation invariant (also called an object invariant) for each class, and pre- and post-conditions for each method. This can be quite a bit of output, so for Assignment 1 we have also supplied an Ant target that reduces the amount of output (and changes a few of the file names). See the Assignment 1 handout for more details.

This target creates three files:

- `daikon.inv.txt`: a text file containing the properties Daikon inferred. This is is the only file you should need to examine.
- `daikon.dtrace.gz`: a trace file of variable values in the program.
- `daikon.inv.gz`: Daikon's output, in binary ("serialized") file. This is useful for running additional tools.

When you run the target, you will see four things:

1. `Executing target program: ...`
   This is output to assist with debugging.
2. The output of the target program itself. For Assignment 1, this is a series of dots indicting successful completion of the JUnit test cases. You will notice that this runs slower than the original test. That is because it is generating a trace file.
3. Progress output from Daikon. This starts with `Executing daikon:` and ends with `Exiting Daikon`.
4. A message indicating that textual results are available in `daikon.inv.txt`.

Serialized (`.inv`) files

Running Daikon both produces textual output (`daikon.inv.txt`) and also outputs the invariants in serialized (binary) form (`daikon.inv.gz`). You can use the `daikon.inv.gz` file to print the invariants:

```
java daikon.PrintInvariants daikon.inv.gz
```

Or, you can insert the invariants as comments in your Java source program (in as many or as few files as you like):

```
java daikon.tools.jtb.Annotate daikon.inv.gz src/hw1/RatNum.java src/hw1/RatPoly.java
src/hw1/RatPolyStack.java
```

The Daikon website has an example of code after invariant insertion.

Understanding the invariants

Program points

Daikon's output is organized in terms of program points, which are locations in the source code. Daikon has four types of program points:

- `foo():::ENTER` is the entry point to method `foo()`, before the method body is executed.
- `foo():::EXIT` is the exit point of method `foo()`. A given method may have many exit points (for example, if it has multiple `return` statements). Daikon's output summarizes facts that are true whenever the procedure is exited.
- `Bar:::OBJECT` is an aggregate program point that combines all method exits and entries. Properties at this program point were true upon entry to and exit from every public method. These properties correspond to the representation invariant (also called the *object invariant*, from which this name comes).
- `Bar:::CLASS` is like `Bar:::OBJECT`, but only for static variables.

Invariant syntax

Daikon uses its own syntax for outputting program properties. Some of the properties will be easy to interpret, but others may be confusing.

The variable `orig(x)` refers to the *original* value of formal parameter `x` - that is, its value on entry to the method. `orig()` variables appear only at `EXIT` program points.

Negative array indices count backwards from the length of the array; for instance, `a[-1]` denotes the last element of array `a`; it is just syntactic sugar for `a[a.length-1]`.

The Daikon manual explains more conventions for variable names and for other aspects of Daikon output.

For problems or questions regarding this page, contact: cse331-staff@cs.washington.edu.