Daikon Invariant Detector Developer Manual

Daikon version 5.8.18

July 22, 2020

Table of Contents

1 Introduction

This is the developer manual for the Daikon invariant detector. See:
http://plse.cs.washington.edu/daikon/.

For information about using Daikon, see the Daikon User Manual. This manual is intended for those who are already familiar with the use of Daikon, but wish to customize or extend it.

Additional information can be found in the technical papers available from http://plse.cs.washington.edu/daikon/pubs/.

2 Extending Daikon

This chapter describes how to customize or modify Daikon.

2.1 Compiling Daikon

To compile Daikon, type `make` in `$DAIKONDIR/java/` or any of its subdirectories. Alternately, type `make -C $DAIKONDIR compile`. To create the `daikon.jar` file, type `make -C $DAIKONDIR daikon.jar`.

The distribution includes `daikon.jar` and compiled `.class` files, so you do not need to compile them yourself unless you make changes.

For more information about compiling Daikon, see the comments in the Makefiles.

2.1.1 Requirements for compiling Daikon

Before compiling Daikon, you need to install some dependencies (that is, software used to build Daikon). For Rocky Linux and Ubuntu, you can find the commands to install these dependencies in file `$DAIKONDIR/scripts/Dockerfile-`*`OSNAME`*`-jdk`*`N`*`-plus`. For other operating systems, use similar commands.

Note that Kvasir, the Daikon front end for the C language (see Section "Kvasir" in *Daikon User Manual*), does not work on Mac OS.

The Daikon build process uses the C preprocessor `cpp` to convert each `.jpp` file in the distribution into multiple `.java` files, which are then compiled. If you have a C compiler, you almost certainly have `cpp`.

2.2 Source code (version control repository)

The Daikon git repository is located on GitHub; see `https://github.com/codespecs/daikon/` .

After making a local clone, see Section 2.1 [Compiling Daikon], page 2, for instructions on how to compile Daikon.

2.3 Using Eclipse

[To be improved.]

Here is one way to use Eclipse to edit Daikon.

First, make sure that Daikon builds cleanly from the command line.

File > Import > General > Existing Projects into Workspace

Choose the `java` directory of your Daikon checkout

Project > properties > Java build path: libraries : `add external jars` everything in the lib/ directory, plus also the `tools.jar` file in the `lib/` directory of your JDK. (I'm not sure why, but `add jars` doesn't show all `.jar` files in the directory.)

Source: add `Daikon`, remove `Daikon/src`. Default output folder: change from `Daikon/bin` to `Daikon`.

2.4 New invariants

You can easily write your own invariants and have Daikon check them, in addition to all the other invariants that are already part of Daikon. Adding a new invariant to Daikon requires writing one Java class, as explained below.

- If you are willing to edit the Daikon source code, you should define the invariant in Daikon's source code, in a subdirectory of `java/daikon/inv/`. Then, edit method `Daikon.setup_proto_invs` to call the new invariant's `get_proto` method and recompile Daikon.
- If you do not wish to edit the Daikon source code, then compile the new invariant and put its `.class` file on your classpath. Then, invoke Daikon with the `--user_defined_invariant` command-line argument (see Section "Options to control invariant detection" in *Daikon User Manual*).

The file `java/daikon/inv/unary/scalar/Positive.java` in the Daikon distribution contains a sample invariant. This invariant is true if the variable is always positive (greater than zero). This invariant is subsumed by other invariants in the system; it is provided only as a pedagogical example. To enable the invariant, comment out the appropriate line in `Daikon.setup_proto_invs()`, then recompile Daikon.

A Java class defining an invariant is a concrete subclass of one of the direct subclasses of `UnaryInvariant`, `BinaryInvariant`, or `TernaryInvariant`. A complete list of invariants appears in the body of `Daikon.setup_proto_invs()`.

Daikon's invariants are first instantiated, then are presented samples (tuples of values for all the variables of interest to the invariant; this might be a 1-tuple, a 2-tuple, or a 3-tuple) in turn. If any sample falsifies the invariant, the invariant destroys itself. All remaining invariants at the end of the program run can be queried for their statistical confidence, then reported as likely to be true.

You need to implement the abstract methods of `Invariant` that are not defined in one of the subclasses listed above. You also need to define a constructor and a static method:

`protected` *`InvName`*`(PptSlice ppt)`
: Constructor for class *InvName*. Should only be called from `instantiate_dyn`. Its typical implementation is

```
super(ppt);
```

`public static` *`InvName`* `get_proto()`
: Returns the prototype invariant used to create other invariants. Its typical implementation is

```
if (proto == null)
  proto = new InvName(null);
return (proto);
```

Methods that need to be overridden that are defined in a subclass of '`Invariant`' include:

`public InvariantStatus check_modified(..., int count)`
`public InvariantStatus add_modified(..., int count)`
: Determines whether the invariant is true for a sample (a tuple of values).

You will eventually want to override one or more of these methods (see Section 2.6 [New formatting for invariants], page 4):

`public String format()`
`public String repr()`
`public String format_using(OutputFormat format)`
: Returns a high-level printed representation of the invariant, for user output.

2.5 New derived variables

A derived variable is an expression that does not appear in the source code as a variable, but that Daikon treats as a variable for purposes of invariant detection. For instance, if there exists an array '`a`' and an integer '`i`', then Daikon introduces the derived variable '`a[i]`'. This permits detection of invariants over this quantity.

(Describing how to create new variety of derived variable is still to be written. For now, see the derived variables that appear in the Java files in directory `$DAIKONDIR/java/daikon/derive/`.)

2.6 New formatting for invariants

Daikon can print invariants in multiple formats (see Section "Invariant syntax" in *Daikon User Manual*).

To support a new output format, you need to do two things:

- In `daikon.inv.Invariant.OutputFormat`, add a new static final field and also update the `get` method.
- In every subclass of `Invariant`, edit the `format_using` method to handle the new `OutputFormat`.

2.7 New front ends

A front end for Daikon converts data into a form Daikon can process, producing files in Daikon's input format — data trace declarations and records. For more information about these files, see Appendix A [File formats], page 31.

The data traces can be obtained from any source. For instance, front ends have been built for stock data, weather forecasts, truck weight data, and spreadsheet data (see Section "convertcsv.pl" in *Daikon User Manual*), among others. More often, users apply a programming language front end (also called an *instrumenter*) to a program, causing executions of the program to write files in Daikon's format. (For information about existing front ends, see Section "Front ends and instrumentation" in *Daikon User Manual*.) When a general front end is not available, it is possible to manually instrument a specific program so that it writes files in Daikon's format. The resulting instrumented program is very similar to what an instrumenter would have created, so this section is relevant to both approaches.

Conceptually, a front end instrumenter has two tasks. Suppose you want to infer invariants at a program point (say, a line of code or the entry or exit from a procedure). The front end must create a declaration (see Section A.3 [Declarations], page 32) that lists the variables in scope at that program point. Every time that program point is executed, the program must output a data trace record (see Section A.4 [Data trace records], page 37). A front end can make the program output a data trace record by inserting a `printf` (or similar) statement that outputs the current values of the variables of interest.

2.7.1 Example instrumented Java program

This section gives an example of how an instrumenter for Java might work; other languages are analogous. Suppose we wish to instrument file `Example.java`.

```
class Example {
  // Return either the square of x or the square of (x+1).
  int squar(int x, boolean b) {
    if (b)
      x++;
    return x*x;
```

```
  }
}
```

The `.decls` file might look like the following.

```
DECLARE
Example.squar:::ENTER
x
int
int
1
b
boolean
int
2

DECLARE
Example.squar:::EXIT
x
int
int
1
b
boolean
int
2
return
int
int
1
```

The instrumented `.java` file might look like the following. This example does not compute the "modified bits", but simply sets them all to 1, which is a safe default.

```
class Example {
  static {
    daikon.chicory.Runtime.setDtraceMaybe("daikon-output/StackAr.dtrace");
  }

  // Return either the square of x or the square of (x+1).
  int squar(int x, boolean b) {
    synchronized (daikon.chicory.Runtime.dtrace) {
      daikon.chicory.Runtime.dtrace.println();
      daikon.chicory.Runtime.dtrace.println("Example.squar:::ENTER");
      daikon.chicory.Runtime.dtrace.println("x");
      daikon.chicory.Runtime.dtrace.println(x);
      daikon.chicory.Runtime.dtrace.println(1);  // modified bit
      daikon.chicory.Runtime.dtrace.println("b");
      daikon.chicory.Runtime.dtrace.println(b ? 1 : 0);
      daikon.chicory.Runtime.dtrace.println(1);  // modified bit
    }

    if (b)
      x++;
```

```
      int daikon_return_value = x*x;
      synchronized (daikon.chicory.Runtime.dtrace) {
        daikon.chicory.Runtime.dtrace.println();
        daikon.chicory.Runtime.dtrace.println("Example.squar:::EXIT");
        daikon.chicory.Runtime.dtrace.println("x");
        daikon.chicory.Runtime.dtrace.println(x);
        daikon.chicory.Runtime.dtrace.println(1);  // modified bit
        daikon.chicory.Runtime.dtrace.println("b");
        daikon.chicory.Runtime.dtrace.println(b ? 1 : 0);
        daikon.chicory.Runtime.dtrace.println(1);  // modified bit
        daikon.chicory.Runtime.dtrace.println("return");
        daikon.chicory.Runtime.dtrace.println(daikon_return_value);
        daikon.chicory.Runtime.dtrace.println(1);  // modified bit
      }

      return daikon_return_value;
    }
  }
```

(Daikon's Java front end, Chicory, does not actually insert instrumentation into the Java source code of your program. Rather, it instruments the bytecode as it is loaded into the JVM. This is more efficient, and it avoids making any changes to your `.java` or `.class` files. We have shown an example of Java source code instrumentation because that is simpler to explain and understand than the bytecode instrumentation.)

2.7.2 Instrumenting C programs

Daikon comes with a front end for the C language: Kvasir (see Section "Kvasir" in *Daikon User Manual*). Kvasir only works under the Linux operating system, and it works only on "x86" (Intel 386, 486, 586, 686 compatible) and "x86-64" (AMD64, EM64T compatible) processors.

You may wish to infer invariants over C programs running on other platforms; for instance, you want a robust C front end that works under Microsoft Windows. This section will help you to either write such a front end or to hand-instrument your program to produce output that Daikon can process.

We welcome additions and corrections to this part of the manual. And, if you write a C instrumenter that might be of use to others, please contribute it back to the Daikon project.

A front end for C (or any other language) performs two tasks. It determines the names of all variables that are in scope at a particular program point, and it prints the values of those variables each time the program point executes.

Determining the names of the variables is straightforward. It requires either parsing source code or parsing a compiled executable. In the latter case, the variables can be determined from debugging information that the compiler places in the executable.

The challenge for C programs is determining the values of variables at execution time: for each variable, the front end must determine whether the variable's value is valid, and how big the value is.

A front end should print only variables that have *valid* values. Examples of invalid values are variables that have not yet been initialized and pointers whose content has been deallocated. (A pointer dereference, such as '`*p`' or '`p->field`', can itself be to uninitialized and/or deallocated memory.) Invalid values should be printed as '`nonsensical`' (see Section A.4 [Data trace records], page 37).

It is desirable to print '`nonsensical`' rather than an invalid value, for two reasons. First, outputting nonsense values can degrade invariant detection; patterns in the valid data may be masked by noise from invalid values. Second, an attempt to access an invalid value can cause the instrumented program to crash! For instance, suppose that pointer '`p`' is not yet initialized — the pointer value refers to some arbitrary location in memory, possibly even an address that the operating system has not allocated to the program. An attempt to print the value of '`*p`' or '`p->field`' will result in a segmentation fault when '`*p`' is accessed. (If you choose never to dereference a pointer while performing instrumentation, then you do not need to worry about invalid references. However, you will be unable to output any fields of a pointer to a struct or class, making your front end less useful. You will still be able to output fields of a regular variable to a struct or class, but most interesting uses of structs and classes in C and C++ are through pointers.)

C relies on the programmer to remember which variables are valid, and the programmer must take care never to access invalid variables. Unfortunately, there is no simple automatic way to determine variable validity for an arbitrary C program. (Languages with automatic memory management, such as Java, do not pose these problems. All variables always have an initial value, so there is no danger of printing uninitialized memory, though the initial value may not be particularly meaningful. Because pointed-to memory is never deallocated, all non-null pointers are always valid, so there is no danger of a segmentation fault.)

An instrumenter needs information about validity of variable values. This could be obtained from the programmer (which requires work on the part of the user of Daikon), or obtained automatically by creating a new run-time system that tracks the information (which requires a more sophisticated front end).

In addition to determining which variables are uninitialized and which pointers are to allocated memory, there are additional problems for a C front end. For example, given a char pointer '`*c`', does it point to a single character, or to an array of characters? If it points to an array of characters, how big is that array? And for each element of the array, is that element initialized or not?

The problem of tracking C memory may seem daunting, but it is not insurmountable. There exist many tools for detecting or debugging memory errors in C, and they need to perform exactly the same memory tracking as a Daikon front end must perform. Therefore, a Daikon front end can use the same well-known techniques, and possibly can even be built on top of such a tool. For instance, one C front end, named Kvasir, is built on top of the Valgrind tool (`https://valgrind.org/`), greatly reducing the implementation effort. Valgrind only works under Linux, but a C front end for another platform could build on a similar tool; many other such tools exist.

There are two basic approaches to instrumenting a C program (or a program in any other language): instrument the source code, or instrument a compiled binary representation of the program. In each case, additional code that tracks all memory allocations, deallocations, writes, and reads must be executed at run time. Which approach is most appropriate for you depends on what tools you use when building your C instrumentation system.

In some cases, it may not be necessary to build a fully general C instrumentation system. You may be able to craft a smaller, simpler extension to an existing program — enabling that program (only) to produce files for Daikon to analyze.

For instance, many programs use specialized memory allocation routines (customized versions of `malloc` and `free`), in order to prevent or detect memory errors. The information that such libraries collect is often sufficient to determine which variable values should be printed, and which should be suppressed in favor of printing '`nonsensical`' instead.

The presence of memory errors — even in a program that *appears* to run correctly — makes it much harder to create Daikon's output. Therefore, as a prerequisite to instrumenting a C program, it is usually a good idea to run a memory checker on that program and to eliminate any memory errors.

2.8 New suppressors

As mentioned in Chapter 4 [Daikon internals], page 12, one way to make Daikon more efficient, and to reduce clutter in the output to the user, is to reduce the number of redundant invariants of various kinds. This section describes how to add a new suppressor relation, such that if invariant A implies B, B is not instantiated or checked as long as A holds, saving time and space. Suppression implications use some terminology. A *suppressor* (defined in the class `NISuppressor`) is one of a set of invariants (`NISuppression`) that imply and suppress a *suppressee* invariant (`NISuppressee`). The set of all of the suppressions that suppress a particular *suppressee* is stored in the class `NISuppressionSet`.

Adding a new suppression is straightforward when the invariants involved do not have any state. Define the *suppressee* and each of the suppressions that suppress it using the corresponding constructors. Add the method `get_ni_suppressions` to the class of the invariant being suppressed and return the appropriate suppression set. Make sure that `get_ni_suppressions` always returns the same suppression set (i.e., that storage to store the suppressions is only allocated once). Normally this is done by defining a static variable to hold the suppression sets and initializing this variable the first time that `get_ni_suppressions` is called.

The following example defines suppressions for '`x == y`' implies '`x >= y`' and '`x > y`' implies '`x >= y`'.

```
private static NISuppressionSet suppressions = null;

public NISuppressionSet get_ni_suppressions() {
  if (suppressions == null) {
    NISuppressee = new NISuppressee(IntGreaterEqual);

    NISuppressor v1_eq_v2 = new NISuppressor(0, 1, IntEqual.class);
    NISuppressor v1_lt_v2 = new NISuppressor(0, 1, IntLessThan.class);

    suppressions = new NISuppressionSet(new NISuppression[] {
      new NISuppression(v1_eq_v2, suppressee),
      new NISuppression(v1_lt_v2, suppressee),
    });
  }
  return suppressions;
}
```

For suppressions depending on the state of a particular invariant, each `Invariant` has an `isObviousDynamically(VarInfo[] vis)` method that is called once the state of other invariants has already been determined. This method returns a non-null value if this invariant is implied by a fact that can be derived from the given `VarInfo`s.

For example, suppose division was not defined for divisors smaller than 1. The following example defines an obvious check for '`x <= c`' (where c < 1 is a constant) implies '`y % x == 0`', written in the Divides class.

```
public DiscardInfo isObviousDynamically(VarInfo[] vis) {
  DiscardInfo di = super.isObviousDynamically(vis);
  if(di != null) {
    return di;
  }

  VarInfo var1 = vis[0];
```

```
    PptSlice1 ppt_over1 = ppt.parent.findSlice(var1);

    if(ppt_over1 == null) {
      return null;
    }

    for(Invariant inv : ppt_over1.invs) {
      if(inv instanceof UpperBound) {
        if(((UpperBound) inv).max() < 1) {
          return new DiscardInfo(this, DiscardCode.obvious,
                          'Divides is obvious when divisor less than one');
        }
      }
    }

    return null;
  }
```

2.9 Reading dtrace files

If you wish to write a program that manipulates a `.dtrace` file, you can use Daikon's built-in mechanisms for parsing `.dtrace` files. (This is easier and less error-prone than writing your own parser.)

You will write a subclass of `FileIO.Processor`, then pass an instance of that class to `FileIO.read_data_trace_files`. Daikon will parse each record in the trace files that you indicate, then will pass the parsed version to methods in your processor.

For a simple example of how to use `FileIO.Processor`, see the file `daikon/java/daikon/tools/ReadTrace.java`.

2.10 System.exit

The Daikon codebase does not call `System.exit()`, except in a dummy main method that catches `TerminationMessage`, which is the standard way that a component of Daikon requests the JVM to shut down.

The reason for this is that calling `System.exit()` is usually a bad idea. It makes the class unusable as a subroutine, because it might kill the calling program. It can cause deadlock. And it can leave data in an inconsistent state (for example, if the program was in the middle of writing a file, still held non-Java locks, etc.), because the program has no good way of completing any actions that it was in the middle of. Therefore, it is better to throw an exception and let the program handle it appropriately. (This is true of instrumentation code as well.)

To see the stack trace for a `TerminationMessage`, pass `--config_option daikon.Debug.show_stack_trace=true` on the command line.

3 Debugging Daikon

Section "Daikon debugging options" in *Daikon User Manual* describes several command-line options that enable logging, which can be a useful alternative to using a debugger when debugging Daikon. Because Daikon processes large amounts of data, using a debugger can be difficult.

This chapter describes some of the command-line options in more detail.

3.1 Track logging

Often it is desirable to print information only about one or more specific invariants. This is distinct from general logging because it concentrates on specific invariant objects rather than a particular class or portion of Daikon. This is referred to as *Track* logging because it tracks particular values across Daikon.

The `--track` *`class|class|...<var,var,var>@ppt`* option to Daikon (see Section "Daikon debugging options" in *Daikon User Manual*) enables track logging. The argument to the `--track` option supplies three pieces of information:

1. The class name of the invariant (e.g., `IntEqual`). Multiple class arguments can be specified separated by pipe symbols ('`|`').
2. The variables that are used in the invariant (e.g., `return`, `size(this.s[])`). The variables are specified in angle brackets ('`<>`').
3. The program point of interest (e.g., `DataStructures.StackAr.findMin()V:::ENTER`). The program point is preceded by an at sign ('`@`').

Each item is optional. For example:

```
IntEqual<x,y>@makeEmpty()
LessThan|GreaterThan<return,orig(y)>@EXIT99
```

Multiple `--track` switches can be specified. The class, program point, and each of the variables must match one of the specifications in order for information concerning the invariant to be printed.

Matching is a simple substring comparison. The specified item must be a substring of the actual item. For instance, `LessThan` matches both `IntLessThan` and `FloatLessThan`.

Program points and variables are specified exactly as they are seen in normal Daikon invariant output. Specifically, `Ppt.name` and `VarInfo.name.name()` are used to generate the names for comparisons.

Invariants are not the only classes that can be tracked. Any class name is a valid entry. Thus, for example, to print information about derived sequence variables from sequence `this.theArray[]` and scalar `x` at program point `DisjSets.find(int):::EXIT`, the tracking argument would be:

```
SequenceScalarSubscriptFactory<x,this.theArray[]>@DisjSets.find(int):::EXIT
```

There are two configuration options that can customize the output. The option `daikon.Debug.showTraceback` will output a stack trace on each log statement. The option `daikon.Debug.logDetail` will cause more detailed (and often voluminous) output to be printed. For more information, see Section "Configuration options" in *Daikon User Manual*.

Note that all interesting information is not necessarily logged. It will often be necessary to add new logging statements for the specific information of interest (see Section 3.1.1 [Adding track logging], page 11). This is covered in the next section.

More detailed information can be found in the Javadoc for `daikon.Debug` and `daikon.inv.Invariant`.

3.1.1 Adding track logging

When you add a new invariant, derived variable, or other component to Daikon, you should ensure that it supports track logging in the same way that existing components do. This section describes how to do so.

Track logging is based around the class name, program point name, and variables of interest. Track logging methods accept these parameters and a string to be printed. `Debug.java` implements the following basic log methods:

```
log (String)
log (Class, Ppt, String)
log (Class, Ppt, VarInfo[], String)
```

The first uses the cached version of the `Class`, `Ppt`, and `VarInfo` that was provided in the constructor. The second uses the specified variables and the `VarInfo` information from `Ppt`. The third specifies each variable explicitly.

When logging is not enabled, calling the logging functions can take a significant amount of time (because the parameters need to be evaluated and passed). To minimize this, a function `logOn()` is provided to see if logging is enabled. It is recommended that code of the following form be used for efficiency:

```
if (Debug.logOn()) {
  Debug.log (getClass(), ppt, "Entering routine foo");
}
```

Track logging also can work with other loggers. Each of the logging methods has an alternative version that also accepts a logger as the first argument. In this case, normal track logging is performed if the class, `ppt`, and vars match. If they don't match, the same information is logged via the specified logger. For example:

```
if (Debug.logOn || logger.isLoggable (Level.FINE)) {
  Debug.log (logger, getClass(), ppt, "Entering routine foo");
}
```

The above will print if either the tracking information matches or if the specified logger is enabled.

Convenience methods are available for track logging invariants. In this case the class name, `ppt`, and variable information are all taken from the invariant. The available methods are:

```
logOn()
logDetail()
log (String)
log (Logger, String)
```

These correspond to the `Debug` methods described above. They are the recommended way to log information concerning invariants.

Track logging also provides one additional level of detail. The function `logDetail()` returns whether or not more detailed information should be printed. This should be used for information which is not normally interesting or especially voluminous output. Often statements using `logDetail()` should be commented out when not in active use.

3.1.2 Track log output

Each call to a track log method will produce output in the same basic format. Space for three variables is always maintained for consistency:

```
daikon.Debug: <class>: <ppt>: <var1>: <var2>: <var3>: <msg>
```

If `showTrackback` is enabled, the '`traceback`' will follow each line of debug output.

4 Daikon internals

This chapter describes some of the techniques used in Daikon to make it efficient in terms of time and space needed. These techniques can be enabled or disabled at the Daikon command line, as described in Section "Running Daikon" in *Daikon User Manual*.

4.1 Avoiding work for redundant invariants

Daikon reduces run time and memory by avoiding performing work for redundant invariants that provide no useful information to the user. There are three basic types of optimization that can be performed for uninteresting invariants: non-instantiation, suppression, and non-printing.

Non-instantiation prevents the creation of an invariant because the invariant's truth value is statically obvious (from the semantics of the programming language), no matter what values may be seen at run time. Two examples are '`A[i] is an element of A[]`' and '`size(A[]) >= 0`'. Non-instantiation is implemented by the by the `isObviousStatically` method. With the equality sets optimization (see Section 4.3 [Equality optimization], page 13), non-instantiation can only happen if all equality permutations are statically obvious. Note that `isObviousStatically` should be used only for invariants that are known to be true. Other code presumes that any statically obvious invariants are true and can be safely presumed when determining if other invariants are redundant.

An invariant can be *suppressed* if it is logically implied by some set of other invariants (referred to as *suppressors*). A suppressed invariant is not instantiated or checked as long as its suppressors hold. For example '`x > y`' implies '`x >= y`'. Suppression has some limitations. It cannot use as suppressors or suppress sample dependent invariants (invariants that adapt themselves to the samples they see and whose equation thus involves a constant such as '`x > 42`'). Suppression also cannot use relationships between variables. For example, it cannot suppress '`x[i] = y[j]`' by '`(x[] = y[]) ~ (i = j)`'. Suppressor invariants can only use variables that are also in the invariant that is being suppressed. In this example, only invariants using the variables '`x[i]`' and '`y[i]`' can be used as a suppressors. See Section 2.8 [New suppressors], page 8 for more information.

Non-printing is a post-pass that throws out any invariants that are implied by other true invariants. It is similar to suppression, but has none of the limitations of suppression. But since it is only run as a post pass, it cannot optimize run time and memory use as suppression can. Non-printing should be used only in cases where suppression cannot. Non-printing is implemented by `ObviousFilter`, which calls the `isObviousDynamically` method on invariants. The `isObviousStatically` method is also used by the non-printing checks; it can be called at the end without reference to equality sets.

More detail can be found in the paper "Efficient incremental algorithms for dynamic detection of likely invariants" by Jeff H. Perkins and Michael D. Ernst, published in Foundations of Software Engineering in 2004; the paper is available from `http://plse.cs.washington.edu/daikon/pubs/invariants-incremental-fse2004-abstract.html`.

4.2 Dataflow hierarchy

The dataflow hierarchy expresses relationships between variables at different program points. It is used to save time and space during invariant generation and to prevent invariants from being printed multiple times.

Suppose there are two program points X and its parent Y. Then every sample seen at X is also seen at Y, and every invariant that is true at Y is also true at X.

Variable z in program point X is related to variable z' in another program point Y by a *flow* relation if every sample seen of z at X is also seen of z' at Y. Y is called a parent program point of X.

For example, all the field variables in the `:::ENTER` program point of a method in class C relate to the field variables in the `:::CLASS` program point of C. This is because the state of C, when in context at the entry `:::ENTER` program point, is also in context at the `:::CLASS` program point. Any invariant that holds true on a parent program point must hold on the child program point. Daikon saves time and space by only computing invariants at the highest parent at which they apply.

Section A.3.2 [Program point declarations], page 33 describe how program points are declared in a Daikon input file. Here we will describe how the `parent` records are typically used to connect program points into a dataflow hierarchy.

Daikon uses three primary relationship types (`PARENT`, `ENTER-EXIT` and `EXIT-EXITNN`) to connect the program points into an acyclic dataflow hierarchy.

- A program point that represents the `ENTRY` or `EXIT` of a static method will have a `parent` record that points to the `CLASS` program point for the containing class.
- A program point that represents the `ENTRY` or `EXIT` of a non-static (instance) method will have a `parent` record that points to the `OBJECT` program point for the containing object.
- An `ENTER-EXIT` edge connects each method's `ENTER` program point with its corresponding `EXCEPTION` and `EXIT` program points.
- An `EXIT-EXITNN` edge connects each method's `EXIT` program point with each of its corresponding `EXIT<id>` program points.

  A program point that represents a `CLASS` will usually not have a `parent` record.
- A program point that represents a `OBJECT` will have a `parent` record that points to the `CLASS` program point for the object if the object has static data members.

When using Daikon, the relations used to describe the dataflow hierarchy may result in some true invariants that are not reported at some program points. However, the invariant will be present in some parent program point. The dataflow hierarchy is used by default, but can be disabled by the `--nohierarchy` flag. When dataflow is enabled, the only samples that are examined by Daikon are the `:::EXIT` program points (plus '`orig`' variables) since these contain a complete view of the data, from which invariants at all other locations can be inferred. For example, Daikon does not need to examine data at `:::ENTER` or `:::OBJECT` program points which are parents of `:::EXIT` in the dataflow hierarchy.

4.3 Equality optimization

When N variables are equal within a program point there will be N(N-1)/2 pairwise invariants to represent the equality within the equal variables, and N copies of every other invariant. For example, if a, b, and c are equal, then '`a == b`', '`a == c`', '`b == c`' will be reported as pairwise invariants, and '`odd(a)`', '`odd(b)`' and '`odd(c)`' will be reported. If the variables will always be equal, then reporting N times the invariants is wasteful. Daikon thus treats equality specially.

Each group of variables that are equal from the start of inferencing are placed in *equality sets*. An equality set can hold an arbitrary number of variables, and replaces the O(N^2) pairwise equality invariants. Every equality set has a leader or *canonical* representation by a variable in the set. Non-equality invariants are only instantiated and checked on the leader. When printing invariants, Daikon reports only invariants on the leader. The user can easily determine that '`odd(a)`' and '`a == b`' imply '`odd(b)`'. Equality optimization can be turned off at the command line with the `--noequality` flag.

5 Testing

Daikon has two sets of tests: unit tests (see Section 5.1 [Unit testing], page 14) and regression tests (see Section 5.2 [Regression tests], page 17). If there are any differences between the expected results and the ones you get, don't check in your changes until you understand which is the desired behavior and possibly update the goals.

The Daikon distribution contains unit tests, but not regression tests (which would make the distribution much larger). The regression tests appear in Daikon's version control repository (see Section 2.2 [Source code (version control repository)], page 2), within the `tests` subdirectory.

5.1 Unit testing

The unit tests are found in `daikon/java/daikon/test/`; they use the '`JUnit`' unit testing framework. They take a few seconds to run. They are automatically run each time you compile Daikon (by running `make` in `$DAIKONDIR/java` or any of its subdirectories). You can also run them explicitly via `make unit`. When you write new code or modify old code, please try to add unit tests.

5.1.1 Invariant format testing

This tests the formatting of invariants with specified input. The tests are configured in the file `InvariantFormatTest.commands` under `daikon/test/`. The `InvariantFormatTest.commands` file must be in the classpath when this tester is run.

The file is formatted as follows:

```
<fully qualified class name> [<instantiate args>]
<type string>
<goal string>+ <- 1 or more goal strings
<sample>* <- 0 or more samples
```

The file format should be the same regardless of blank or commented lines except in the samples area. No blank lines or comments should appear after the goal string before the first sample or between parts of samples (these lines are used to determine where sample lists end). This will be remedied in a future version of the tester.

`Instantiate args`
: These are optional additional arguments to the static instantiate method of the class. Each argument consists of the type (boolean or int) followed by the value. For example:

```
boolean true
int 37 boolean false
```

`Type string:`
: A type string must consist of one or more of the following literals: '`int`', '`double`', '`string`', '`int_array`', '`double_array`', or '`string_array`', separated by spaces. This string represents the types that an invariant is supposed to compare For instance, a binary integer comparison would have type string '`int int`'. A pairwise sequence comparison would have type string '`int_array int_array`'.

`Goal string:`
: The goal string must start with the prefix '`Goal `', and then continue with '`(<format type>): `', where format type is the format in which the invariant will print. After this the representation of the invariant must occur. It must represent the invariant result exactly as printed, even white space is significant (as proper formatting should be correct down to the whitespace). The first variable (the one corresponding to the

first type in the type string) corresponds with 'a', the second with 'b' and so on. Format the type string accordingly. (In samples, the value of 'a' is read first, possibly followed by 'b', and then possibly 'c', depending on the arity of the invariant.)

```
Example:
Type string, Goals
 |             |
\|/            |
int           \|/
Goal (daikon): a >= -6
Goal (java): a >= -6
Goal (esc): a >= -6
Goal (ioa): a >= -6
Goal (jml): a >= -6
Goal (simplify): (>= |a| -6)
```

Note that the spacing on the goal lines is exact, that is, no extra spaces are allowed and no spaces are allowed to be missing. So the exact format is again:

```
Goal<1 space>(<format name>):<1 space><goal text>
```

`Samples:` Values formatted according to the type string, one value per line Make sure that the samples provided are actually instances of that particular invariant (That is, if the desired invariant is 'a < b', then the first number of each sample better be less than the second)

Arrays and strings must be formatted according to the Daikon `.dtrace` file convention (for a full description, see Appendix A [File formats], page 31). This states that arrays must be surrounded in brackets (start with '[', end with ']'), and entries must be separated by a space. Strings must be enclosed in quotes ('"'). Quotes within a string can be represented by the sequence '\"'.

For example:

```
[1 2 3 4 5] - an array with the elements 1, 2, 3, 4, 5
"aString" - a string
"a string" - also legal as a string
"\"" - the string with the value "
["a" "b" "c"] - an array of strings

int int        <- type string
Goal: a < b    <- goal string, no comment/blank lines after this
1              <- or before this
2
2              <-|__ Pair of values (a = 2 , b = 3)
3              <-|
```

Other examples are in the existing test file (`InvariantFormatTest.commands`).

The output of a test run can be converted into goals by using the `--generate_goals` switch to the tester as follows:

```
java daikon.test.InvariantFormatTester --generate_goals
```

Note that this test is included in the set of tests performed by the master tester, and so it is not necessary to separately run this test except to generate goal files.

Furthermore, this framework cannot parse complex types from files unless they contain a `public Object valueOf(String s)` function. Otherwise the program has no was of knowing how to create such an object from a string. All primitives and the String type are already recognized.

5.1.2 Sample Testing

Sample testing tests various components of Daikon as samples are being processed. A file (normally `daikon/test/SampleTester.commands`) specifies a `.decls` file to use, the samples for each '`ppt/var`', and assertions about Daikon's state (such as whether or not a particular invariant exists).

Each line of the file specifies exactly one command. Blank lines and leading blanks are ignored. Comments begin with the number sign ('`#`') and extend to the end of the line. The type of command is specified as the first token on the line followed by a colon. The supported commands are:

`decl:` *decl-file* [SampleTester Command]
This command specifies the declaration file to use. This is a normal `.decls` file that should follow the format defined in the user manual.

`ppt:` *ppt* [SampleTester Command]
This command specifies the program point that will be used with following vars, data, and assert commands. The program point should be specified exactly as it appears in the `.decls` file.

`vars:` *var1 var2...* [SampleTester Command]
Specifies the variables that will be used on following data lines. Each variable must match exactly a variable in the `ppt`. Other variables will be treated as missing.

`data:` *val1 val2...* [SampleTester Command]
Specifies the values for each of the previously specified variables. The values must match the type of the variables. A single dash (-) indicates that a variable is missing.

`assert:` *assertion* [SampleTester Command]
Specifies an assertion that should be true at this point (see Section 5.1.2.1 [Assertions], page 16). The negation of an assertion can be specified by adding an exclamation point before the assertion (for example: `!inv("x > y", x, y)`).

5.1.2.1 Assertions

Assertions are formatted like function calls: `<name>(arg1, arg2, ...)`. The valid assertions for the assert: command are:

`inv` *format var1 ...* [Assertion]
The `inv` assertion asserts that the specified invariant exists in the current `ppt`. The *format* argument is the result of calling `format()` on the invariant. This is how the invariant is recognized. The remaining arguments are the variables that make up the invariants slice. These must match exactly variables in the `ppt`. The `inv` assertion returns true if and only if the slice exists and an invariant is found within that slice that matches *format*.

Optionally, *format* can be replaced by the fully qualified class name of the invariant. In this case, it is only necessary for the class to match.

More assertions can easily be added to `SampleTester.java` as required.

5.1.2.2 Example file

The following is an simple example of sample testing.

```
decl: daikon/test/SampleTesters.decls

ppt: foo.f():::EXIT35
  vars: x y z
```

```
data: 1 1 0
data: 2 1 0
assert: inv("x >= y", x, y)
assert: inv(daikon.inv.binary.twoScalar.IntGreaterEqual,x,y)
assert: !inv("x <= y", x, y)
```

5.2 Regression tests

The regression tests run Daikon on many different inputs and compare Daikon's output to expected output. They take about an hour to run.

The regression tests appear in the `$DAIKONDIR/tests/` directory. Type `make` in that directory to see a list of Makefile targets. The most common target is `make diffs`; if any output file has non-zero size, the tests fail. You do not generally need to do `make clean`, which forces re-instrumentation (a possibly slow process) the next time you run the tests.

As when you install or compile Daikon, when you run the tests environment variable `DAIKONDIR` should be set.

You should generally run the regression tests before checking it a change (especially any non-trivial change). If any of the regression test diffs has a non-zero size, then your edits have changed Daikon's output and you should not check in without carefully determining that the changes are intentional and desirable (in which case you should update the goal output files, so that the diffs are again zero).

There are several subdirectories under `$DAIKONDIR/tests/`, testing different components of the Daikon distribution (such as Kvasir, see Section "Kvasir" in *Daikon User Manual*). Tests of the invariant detection engine itself appear in `$DAIKONDIR/tests/daikon-tests/`.

Each Makefile under `$DAIKONDIR/tests` includes `$DAIKONDIR/tests/Makefile.common`, which contains the logic for all of the tests. `Makefile.common` is somewhat complicated, if only because it controls so many types of tests.

5.2.1 Kvasir regression tests

The Kvasir (Daikon C front-end) tests appear in the `$DAIKONDIR/tests/kvasir-tests` directory. These tests run Daikon to ensure that the Kvasir output is valid Daikon input. To run them, go to `$DAIKONDIR/tests/kvasir-tests` or of its sub-directories and run `make summary-w-daikon`. If any tests return '`FAILED`', then you should look at the appropriate `.diff` file. If you feel that the failure was actually a result of your Daikon changes and should be in fact correct output, then run `make update-inv-goals` to update the Daikon `invs.goal` file.

5.2.2 Adding regression tests

Most Daikon regression tests are in subdirectories of `$DAIKONDIR/tests/daikon-tests/`. (There is also a `$DAIKONDIR/tests/chicory-tests/` directory, but it is usually better to put tests in `$DAIKONDIR/tests/daikon-tests/`, even if they are exercising Chicory-specific behavior.)

To create a new test, perform the following steps.

1. Create a new subdirectory of `$DAIKONDIR/tests/daikon-tests/`.
2. Put the source files for the test under `$DAIKONDIR/tests/sources/`, not in the test directory itself. Your test may be able to re-use existing Java source code that appears in that directory.
3. It is expedient to copy a Makefile from another subdirectory, such as `$DAIKONDIR/tests/daikon-tests/StackAr/`, then modify it.

   The `Makefile` must contain at least the following entries.

'`MAIN_CLASS`'
: Dot separated fully qualified name of the class that contains the main entry point for the test. For example,

```
MAIN_CLASS := DataStructures.StackArTester.
```

'`include ../../Makefile.common`'
: This includes the common portion of the test Makefiles that does most of the work. See it for more information on the details of regression testing.

'`instrument-files-revise:`'
: A target that writes the list of files to instrument. For example,

```
instrument-files-revise:
    echo "DataStructures/StackAr.java" >| ${INST_LIST_FILE}
```

If you run `make` (without a target), you will see a description of all of the Makefile's functionality. Most of that is inherited from `$DAIKONDIR/tests/Makefile.common`.

4. The `.goal` files are the expected results of running Daikon and its associated tools.

   For example, the `StackAr` directory contains the following `.goal` files, among others:

```
Makefile
Stackar.spinfo-static.goal
StackAr.txt-daikon.goal
StackAr.txt-esc.goal
StackAr.txt-jml.goal
StackAr.txt-merge-esc.goal
StackAr.txt-merge-jml.goal
```

   If you omit some `.goal` files, then the related tests will not be run. (If you can't figure out how to ensure the tests are not run, it may be easier to just add the additional `.goal` files.)

   You can start out with the `.goal` files empty. Execute `make diffs`, to produce output; the tests will fail because the output is not identical to the empty `.goal` files. When the test output is satisfactory, execute `make update-goals` to copy the actual results to the `.goal` files. Then, commit the goal files, Makefile, and source files to the repository.

5. Make the new test be run by adding to the appropriate list (usually '`everything`' or '`quick`') in `$DAIKONDIR/tests/daikon-tests/Makefile`.

For more information, see the comments in file `$DAIKONDIR/tests/Makefile.common`.

6 Editing Daikon source code

The Daikon source code follows the Google Java Style and also Michael Ernst's coding conventions.

6.1 Eclipse setup

6.1.1 Start up Eclipse

Just run the command `eclipse` at the command shell.

There is one big catch: take note of where you launch Eclipse and **always** launch Eclipse from the same location. For example, you might want to alias run-eclipse with `cd $HOME; eclipse`. For convenience, this is already done for you if you source the `daikon-dev.bashrc` startup script. There are ways around the launch location inconvenience, such as the -data switch, but the alias trick should be the simplest and least confusing workaround.

It is strongly recommended that you make the following customizations to Eclipse:

```
Window >> Preferences >> Workbench >> Refresh Workspace on Startup
Window >> Preferences >> Java >> Compiler >> Problems >> Unused imports >> Ignore
Window >> Preferences >> Java >> Compiler >> Style >> Non-static access >> >> Ignore
```

6.1.2 Work on existing coding projects

6.1.2.1 Import an existing project

Second, import Daikon into Eclipse. To import: start Eclipse, then choose menu item '`File->Import->ExistingProjectIntoWorkspace`', then enter (the full name of) directory `$DAIKONDIR/java` for Project Contents.

Those two steps are all there is to it!

(This simple procedure works because there are already `.project` and `.classpath` files in `$DAIKONDIR/java`. To import a project that doesn't have these files, you would need to create them (for instance, copy them from Daikon's), or else follow a different procedure to work on your project in Eclipse.)

6.1.2.2 Importing a different existing project

Before working on code in Eclipse, you must import the code into the Eclipse workspace. You also need to set up various compilation settings and specify jars and classes that should be in the compilation classpath. (By default, Eclipse does not use your `CLASSPATH` environment variable.)

1. Change to the Java perspective by selecting `"Java Perspective"` in the perspective menu on the far-left tool panel. You should see some of the windows move around to a structure similar to the screenshot below.
2. Select the menu item File->New->Project
3. On the popup window, select Java Project and click Next
4. Call the project '`Daikon`' or some other appropriate name and click Next.
5. On the next window, create a new folder in the project called `src`, and answer Yes to the next dialog box asking you to confirm a change in the build directory. Select '`Ok`' and you should see the new Project as a small folder icon in the package explorer.
6. Select the menu item File->Import
7. On the next popup window, select '`File System`' and click Next. Browse to the folder containing your source code. Using the standard '`PAG`' setup, the folder should be `$DAIKONDIR/java`.

The import destination should be `Daikon/src`. Check the boxes for including Daikon, '`jtb`', and plume.

8. Finally, add the required jars for Daikon by right-clicking on the project and selecting the Properties menu. Select "Java Build Path", then "Libraries", then "Add External Jars" and add `tools.jar` and all `.jar` files in `daikon/java/lib/`.

Note that if you follow these directions, then **the compiled classes do not appear in the directory from which you imported in the previous step**. Instead, the compiled classes appear in the Eclipse workspace directory. If you followed the setup instructions above, then the compiled code will be in "$HOME/.eclipse/Daikon/bin", which you should add to your classpath in order to use the updates you make from Eclipse.

6.1.3 Interaction with external tools

Eclipse compiles your project whenever you save. However, it parses frequently during editing and issues most or all of the errors and warnings that the compiler would have. In order to clear a warning/error (in the Task List), you must save the file, however.

If you change files outside Eclipse, you should refresh Eclipse, via right-clicking on the project or via menu item File >> Refresh.

6.1.4 Using the Eclipse debugger

To begin debugging, you can click on the bug icon in the toolbar or select Run->Debug from the menu. You will see the same window as if you selected Run->Run, but when the Java application launches, Eclipse will switch to the Debugging Perspective, which reveals many windows such as the stack trace window, the variable values window, and the step control panel.

Debugging features (window locations given for default setup):

- Breakpoints: You can add breakpoints during editing by double-clicking on the left margin area of the source code. When debugging, the program halts when it reaches a breakpoint.
- Stack Trace: Once the program halts at a breakpoint, the window in the upper-left corner window show the stack trace.
- Step Controls: Once the program halts at a breakpoint, the panels just below the stack trace allow you to step through the code one line at a time (step over), resume execution as normal (resume), step down into a method call (step into), or execute until leaving the current method (Step return).
- Variables: On the top right window, there is a tab pane that displays the values of objects and fields during a breakpoint halt. You can also type in any legal Java expression that could appear at the breakpoint, and the window will display the value of the expression (don't forget to put a semi-colon at the end of your expression!).

6.1.5 Using Git with Eclipse

This section has yet to be written, although you can just do all the editing, compiling, and running from the Eclipse IDE while doing version control commands from the shell or whatever current system you use.

6.2 Editing daikon.texinfo

The Daikon manual appears in `doc/daikon.texinfo`. The manual is written using the Texinfo formatting system, which makes it easy to produce HTML, info, or printed (PDF) output.

You can edit the manual using an ordinary text editor. Emacs has a useful Texinfo mode. You can mimic the formatting commands that already appear in the manual, but it's also a good idea to read the Texinfo documentation.

If you wish to create a new section of the manual, insert two lines like the following:

```
@node    @Short title
@chapter @Long title
```

where you select the short and the long titles (which may be the same), and you may replace `@chapter` by `@section`, `@subsection`, or `@subsubsection`.

Follow the Texinfo conventions for marking up text. In particular:

- Don't use regular double quotes (`"`) in running text. Instead, use two single back quotes and two regular quotes, "like this". Emacs will do this substitution for you if you are using its Texinfo mode.
- Often, some other kind of markup is better than quotes:
  - `@code` is for names of classes, functions, variables, keywords, and other fragments of source code.
  - `@var` is for meta-syntactic variables, not for referring to variables that appear literally in the source code.
  - `@file` is for the names of files.
  - `@option` is for command-line options.
  - `@samp` is for sample input or output. (Alternatively, you can use `@example` to set off longer input/output strings.)
  - `@command` is for command lines.
- Examples shouldn't include extra spaces on the left for indentation since Texinfo already adds some indentation. Remember that in code examples you need to replace all curly braces with { and } to quote them for Texinfo.
- Don't mix `@example` and `@smallexample` in the same section since it looks funny to have examples in two different sized fonts. If your example is too wide to fit on the page with `@example`, it's better to insert some line breaks to make it more readable than to shrink the font.

Once you have edited the manual, run `make` twice (yes, you must run it twice) in order to generate formatted versions of the manual (`daikon.{html,pdf}`).

7 Making a distribution (making a release)

This section provides the instructions for publishing a Daikon distribution, or making a release. If you only want to create the `daikon.tar` or `daikon.tar.gz` file in your own directory, then simply run `make daikon.tar` or `make daikon.tar.gz`.

Official releases have even version numbers (e.g., 4.6.4) and intermediate work has odd version numbers (e.g., 4.7.3). This means as you prepare for a release the current version number is probably odd. It will be updated as one of the steps in the release process. After making the distribution, one of the final steps is to increment the version number again to prepare for subsequent development. This system has the useful side effect of allowing the build and test process to be repeated to fix a problem without having to worry about updating or resetting the version number. Another advantage is to reinforce, to people who are working from the version control repository, that they are not using the released version, because the version numbers differ.

The Daikon distribution site is located at http://plse.cs.washington.edu/daikon/ and is served from the UW CSE file system at `/cse/web/research/plse/daikon`. In order to be able to write to the distribution site, your CSE user id must be a member of the '`plse_www`' Unix group.

For each of the major steps below, an approximate elapsed time is listed. These timings are up to date as of December 2015. They were measured on a quad x86-64 based machine at 3.4GHz with 16GB of memory (buffalo.cs.washington.edu). Barring any difficulties, the entire process will take at least two hours — but could be much more depending on the number of different platforms on which you test the release.

Each of the steps below assumes that you are using the Bash shell.

7.1 Directory layout requirements

- Environment variable `DAIKONDIR` must be set to a clone of https://github.com/codespecs/daikon.
- A clone of https://github.com/codespecs/fjalar must be a sibling of `$DAIKONDIR`.
- Environment variable `JAVA_HOME` must be set to the appropriate JDK installation. For now, this should be a current version of JDK 8. If the default JDK on your system is not JDK 8, you will also need to add `$JAVA_HOME/bin` to your `PATH`. See the example in the next section.
- Optionally, set environment variable `BIBDIR` to a clone of https://github.com/mernst/plume-bib.

7.2 Distribution setup instructions

Get a fresh copy of your Bash shell (e.g., log out and log back in), then run the following (adjust environment variables as necessary):

```
export DAIKONDIR=$inv/daikon
export JAVA_HOME=/usr/lib/jvm/java-1.8.0
export PATH=$JAVA_HOME/bin:$PATH
unset CHECKERFRAMEWORK
```

7.3 Getting the latest version of dependencies

Update the Daikon source files to their most recent version.

```
set -o pipefail
(cd $DAIKONDIR && git pull && git log --branches --not --remotes && git status)
(cd $DAIKONDIR/../fjalar && git pull && git log --branches --not --remotes && git status)
```

Each of the two commands should print exactly these lines:

```
Already up-to-date.
On branch master
Your branch is up-to-date with 'origin/master'.

nothing to commit, working directory clean
```

[Time: moments]

The Fjalar tool set (primarily, Kvasir) is built upon, or uses pieces from, two open source projects. The home page for the Valgrind instrumentation framework is https://valgrind.org. The home page for the GNU Binutils (a collection of binary tools, of which Kvasir uses only readelf) is http://www.gnu.org/software/binutils/.

File $DAIKONDIR/fjalar/valgrind/REVISION indicates the version of these tools that Kvasir uses. You can determine whether a newer version of these tools is available by comparing the REVISION file to https://valgrind.org/downloads/current.html and http://ftp.gnu.org/gnu/binutils/?C=M;O=D. If so, you should update the Fjalar source tree as soon as practical. For details, see the separate document "Merging newer versions of Valgrind into Fjalar", which appears in the fjalar repository.

7.4 The day before the release

Do these steps the day before the release, so that tests have time to complete overnight.

- Edit the doc/CHANGELOG.md file to indicate what has changed in this version. A good way to determine what has changed is to diff the Texinfo source files in the doc/ directory against the previous versions:

  ```
  cd $DAIKONDIR/doc
  diff -b -u -s --from-file /cse/web/research/plse/daikon/download/doc *.texinfo
  ```

  Another good source of information are the repository logs since the last release:

  ```
  cd $DAIKONDIR
  DAIKONVERSION=`wget -q http://plse.cs.washington.edu/daikon/download/doc/VERSION \
      -O - | xargs echo -n`
  git log v$DAIKONVERSION..HEAD
  ```

  In addition, you should run the same command in your Fjalar clone:

  ```
  cd $DAIKONDIR/fjalar
  DAIKONVERSION=`wget -q http://plse.cs.washington.edu/daikon/download/doc/VERSION \
      -O - | xargs echo -n`
  git log v$DAIKONVERSION..HEAD
  ```

  When you have competed your updates, commit and push the changes.

  [Time: a few minutes]

- Check the release documents for spelling errors.

  ```
  cd $DAIKONDIR/doc
  make spell-check
  ```

  Any words that may be spelled incorrectly are output to standard out.

  - If the word is misspelled, correct it in the source (.texinfo) file.
  - If the word is spelled correctly and is a normal English word, add it to daikon.dict, keeping the list in alphabetical order.
  - If the word is spelled correctly and is a proper name, such as a person's or company's name, then either add it to daikon.dict or write @nospellcheck around it.
  - If the word is technical information, such as part of a computer command or filename, then mark it with a Texinfo command; see "Chapter 9: Marking Text, Words and Phrases" of the GNU Texinfo Manual (PDF, HTML).

[Time: moments]

- Update dependencies in `daikon/java/lib/`. For details, see `daikon/java/lib/README`.

TODO: it would be good to build the staging release the day before, too, so that links can be fixed the day before the release or at least very early in the process.

7.5 Steps for making the distribution

1. Verify that Daikon passes its tests.

   All of the jobs at https://dev.azure.com/mernstdaikon/daikon/_build should pass (they should be green). Here is a list of them:

   https://dev.azure.com/mernstdaikon/daikon/_build/latest?definitionId=1&branchName=master

   https://dev.azure.com/mernstdaikon/daikon/_build/latest?definitionId=2&branchName=master

   If any of the jobs is not passing, then correct the problem and wait for the jobs to complete and pass. The delay to wait for this to happen is a reason that you should avoid making changes to Daikon on the release day. Instead, you should make changes the day before to permit the continuous integration jobs to run overnight.

2. Do a very clean rebuild of everything.

   ```
   cd $DAIKONDIR && make very-clean clean-kvasir && make rebuild-everything
   ```

   [Time: 20 min]

3. Once you have successfully reached this point, you are ready to make a release candidate and begin testing it. The following command will verify that `doc/CHANGELOG.md` is up-to-date and then list any uncommitted files.

   ```
   make check-repo
   ```

   The result of the command above should be:

   ```
   On branch master
   Your branch is up-to-date with 'origin/master'.
   nothing to commit (use -u to show untracked files)
   ```

   If the command output lists any files, they need to be committed to the repository and pushed now. In that case, you will need to return to step 1 and wait for the CI jobs to complete.

   [Time: moments]

4. Set the version number to an even number. You may:

   - Manually edit file `$DAIKONDIR/doc/VERSION`, or
   - If the revision number just needs to be incremented by 1 (to go from odd to even) you may use:

     ```
     make update-dist-version-file
     ```

   Note that if a new feature has been added, or if some change has made the current version incompatible from the previous release (such as a change in the dtrace file format or revised names for tool options), then you should manually edit the VERSION file to increment the minor version number and reset the revision number to zero.

   Optionally, override the distribution date (default: today) by redefining the environment variable `TODAY`:

   ```
   export TODAY='November 1, 2013'
   ```

Now, update the appropriate files with the date and version number of the release and commit these changes back to the repository:

```
make update-run-google-java-format && make update-and-commit-version
```

[Time: moments]

5. It is important to build the release candidate from a clean enlistment. (TODO: In the future, perhaps do all release-related work on a branch. Otherwise, use of the below commands requires committing changes to master, and there might be many commits if there is trouble with the release.) This is to ensure we are testing only the files intended for release and that those files are the same as a user would receive with a clone of the repository. This needs to be done in a separate, new Bash command shell. The following commands get a fresh clone of Daikon, set some environment variables and then build everything to be distributed and copy it to the temporary staging area (http://plse.cs.washington.edu/staging-daikon/).

   ```
   # First, start a separate, new Bash command shell.  Don't kill your old one.

   rm -rf /tmp/$USER/stage-daikon
   mkdir -p /tmp/$USER/stage-daikon
   cd /tmp/$USER/stage-daikon
   git clone --depth 3 https://github.com/codespecs/daikon
   cd daikon
   export DAIKONDIR=`pwd`
   export JAVA_HOME=/usr/lib/jvm/java-1.8.0
   export PATH=$JAVA_HOME/bin:$PATH
   unset CHECKERFRAMEWORK
   make compile && make staging
   ```

   The final output of this command will be a list of files that were added/removed since the last release. If any of these differences is unexpected, then investigate. If any corrections are required, do so back in the main repository, commit the changes, and then repeat this step.

   [Time: 23 min]

6. Next, check the staging website to see if it has any broken document links.

   ```
   make check-for-broken-doc-links
   ```

   Review `checklink.log` and make corrections as appropriate.

   Sometimes a website forbids robots, or artificially slows them. This can lead to link-checking failures. After you manually check a URL for correctness, you can add it to https://github.com/plume-lib/html-tools/blob/master/html-valid-urls so that no warning is issued in the future.

   In some cases a "403 Forbidden" error is transient, due to the website being down. You can check such links by hand.

   If the URL is correct but cannot be checked (for example, because the website prohibits spiders or because the URL redirects to another URL but you prefer to keep the first URL in your document), then you may need to add lines to an appropriate part of file `checklink-args.txt` in https://github.com/plume-lib/checklink.

   In most cases (and for most 404 errors), you should fix the document with the incorrect link. Here is a workflow that may be used to deal with broken links:

   - Use the '`Wayback Machine`', http://archive.org/web/, and enter the URL.
   - You will get a timeline. Select a year and day, then you have to click on the time of day, and then you get a version of the page at that time.
   - Copy the URL from the address bar of your browser.

- If the web search is not successful, email the maintainer of the old content.

Note that you must fix problems in the original repository, not in the drop directories. This means:

- Switch back to your original Bash shell.
- Correct the file. (Most likely located beneath either `$DAIKONDIR/doc` or `$BIBDIR` (`~/bib/` if `$BIBDIR` is not set).)
- Commit the changes.
- Repeat the previous step to rebuild the release candidate.
- Repeat this step to verify your corrections.

If `checklink.log` contains an error message of the form:

```
List of broken links and other issues:
http://plse.cs.washington.edu/daikon/download/doc/developer/Requirements-for-compiling-Kvasir.html
  Line: 185
  Code: 200 OK
 To do: Some of the links to this resource point to broken URI fragments
        (such as index.html#fragment).
The following fragments need to be fixed:
        Requirements-for-compiling-Kvasir       Line: 185
```

This is probably caused by the way `makeinfo` deals with chapter splits via indirect references through stub files. Two `suppress-fragment` entries in `checklink-args.txt` in https://github.com/plume-lib/checklink are required. For the example above, these would be:

```
--suppress-fragment http://plse.cs.washington.edu/daikon/download/doc/developer/\
Requirements-for-compiling-Kvasir.html#Requirements-for-compiling-Kvasir
--suppress-fragment http://plse.cs.washington.edu/staging-daikon/download/doc/developer/\
Requirements-for-compiling-Kvasir.html#Requirements-for-compiling-Kvasir
```

but on two long lines without any `\` line continuation characters.

[Time: 8 min]

7. Test the staged distribution on a 64-bit Rocky Linux machine. Since the machine you are using to build the release is probably such a machine, you may simply run the command below. Success is indicated by invariant output being written to the screen.

   ```
   make test-staged-dist
   ```

   [Time: 5 min]

8. Test the distribution on Ubuntu and Mac OS machines.

   To use Azure Pipelines, navigate to https://dev.azure.com/mernstdaikon/daikon/_build?definitionId=3&_a=summary and click "Run pipeline" then "Run".

   If there is a problem, fix it and start over.

   [Time: 15 min, or longer if the CI service is busy running other jobs for the codespecs organization.]

9. Once you have successfully reached this point, you have a valid release candidate and are ready to make the actual release. **Caution:** If somewhere above you made a change to correct a problem you should have restarted the release process.

   At this point we are done building and testing the release candidate and you should exit the Bash command shell created in step 5 and return to your original Bash shell.

10. Update the website by deleting the current release and then making the staged version the (new) current release.

    ```
    cd $DAIKONDIR
    make staging-to-www
    ```

[Time: 1 min]

11. Add a version label to the repository:

```
cd $DAIKONDIR
DAIKONVERSION=`cat $DAIKONDIR/doc/VERSION | xargs echo -n`
git tag -a v$DAIKONVERSION -m $DAIKONVERSION
git push --tags
cd fjalar
git tag -a v$DAIKONVERSION -m $DAIKONVERSION
git push --tags
```

[Time: moments]

12. At this point the distribution has been completed. Bump the version number to an odd value for continuing development.

```
cd $DAIKONDIR
export -n TODAY
make update-dist-version-file
make update-doc-dist-date-and-version
```

[Time: moments]

13. Verify these changes by doing product builds (no need to clean first) and running a short verification test.

```
cd $DAIKONDIR
make rebuild-everything quick-test
```

Success is indicated by invariant output being written to the screen.

[Time: 5 min]

14. Commit changes back to the repository:

```
cd $DAIKONDIR
git commit -a -m "Bump version number for ongoing development."
git push
cd fjalar
git commit -a -m "Bump version number for ongoing development."
git push
```

[Time: moments]

15. Send mail to the '`daikon-announce`' mailing list. Use the suggested template below, replacing <version number> with the actual number of the release.

```
<to:>  daikon-announce@googlegroups.com
<subject:>  Daikon version <version number> has been released

Daikon version <version number> is available at:
http://plse.cs.washington.edu/daikon/download/

Please see the entry from the doc/CHANGELOG.md file that appears below for more details.

  <your name>

<a copy of the current entry from the doc/CHANGELOG.md file, with
paragraphs refilled (remove unnecessary line breaks that trim lines
to 80 columns for the CHANGELOG.md file but aren't desirable in email)>
```

Note that if you use Microsoft Outlook 2010 or 2013 as your mailer, by default it will insert hard breaks in your outgoing message. See https://blog.

techhit . com / 551102-how-to-prevent-outlook-from-adding-line-breaks-to-sent-plain-text-messages for a work around. You must quit and restart Outlook to activate the change.

8 Analyzing historical versions of Daikon

Daikon's version control repository, available at `https://github.com/codespecs/daikon`, contains complete development history starting from October 1, 2010, and partial development history before that time. The reason for this is that converting the full CVS repository would create a repository that would be too large for convenient use, and that would also be too large to upload to Google Code or GitHub. The original CVS repository is available for download from `http://plse.cs.washington.edu/daikon/download/inv-cvs/`, as files `inv-cvsaa` through `inv-cvsai`. Download all 9 files, and then run the following commands to re-create the CVS repository, which will be named `invariants/`:

```
cat inv-cvsa? > inv-cvs-full.zip
unzip inv-cvs-full.zip
```

Developers can usually make do with the Git repository, which has complete history for the Java source code.

The CVS repository is most often used by researchers — for example, in the testing community — because it contains both a CVS repository and a set of tests that is more extensive than those in the Daikon distribution. If you are a researcher who uses the Daikon version control history, please let us know of any publications so that we can publicize them at `http://plse.cs.washington.edu/daikon/pubs/#daikon-testsubject`.

The layout of the Daikon CVS repository differs slightly from that of the distribution. For example, the top-level directory is named `invariants/` instead of `daikon/`.

The remainder of this section points out some pitfalls for such researchers. Although these problems are easy to avoid, some previous published work has made these mistakes; don't let that happen to you!

Recall that Daikon contains two sets of tests (see Chapter 5 [Testing], page 14); you should include both in any analysis of Daikon's tests. (Or, if you can analyze only one of the two sets of tests, then clearly explain that the regression tests are the main tests.) The regression tests use Makefiles to avoid re-doing unnecessary work, so any description of the time taken to run Daikon's tests should be a measurement of re-running the tests after they have been run once, not running them from a clean checkout or after a `make clean` command.

Daikon intentionally does not contain tests for third-party libraries that are included (sometimes in source form) in the Daikon distribution. As one example, the `java/jtb/` directory contains an external library. Therefore, any measurement of Daikon's code coverage should not include such libraries, whether distributed in source or as `.jar` files.

Historically, the file `doc/www/mit/index.html` in the CVS repository contains information about how Daikon's developers use Daikon. This file changes from time to time — for instance, it changed when a CVS branch was created and later when development on it ceased (see Section 8.1 [Branches], page 29).

8.1 Branches

The Daikon CVS repository contains two branches: a main trunk and a branch (named '`ENGINE_V2_PATCHES`') for version 2 of Daikon.

The CVS manual (see section "Branching and merging" of the manual *CVS — Concurrent Versions System*) describes CVS branches:

> CVS allows you to isolate changes onto a separate line of development, known as a *branch*. When you change files on a branch, those changes do not appear on the main trunk or other branches.

> Later you can move changes from one branch to another branch (or the main trunk) by *merging*. Merging involves first running `cvs update -j`, to merge the changes into the working directory. You can then commit that revision, and thus effectively copy the changes onto another branch.

In early January 2002 (or perhaps in late 2001), we created the '`ENGINE_V2_PATCHES`' branch at the `invariants/java/daikon level` of the Daikon CVS repository. Primary development continued along the CVS branch '`ENGINE_V2_PATCHES`', which we called "Daikon version 2" We called the CVS trunk "Daikon version 3" it was experimental, and very few people ran its code or performed development on it. Periodically, all changes made to the branch would be merged into the trunk, as one large checkin on the trunk. Later, development on version 3 became more common, some changes were merged from the trunk to the branch, and version 2 was finally retired (and no more changes were made to the branch) in December 2003.

A regular `cvs checkout` gets the trunk. The `-r` flag specifies a branch. For example, to get the branch as of June 9, 2002, one could do

```
cvs -d $pag/projects/invariants/.CVS co -r ENGINE_V2_PATCHES \
    -D 2003/06/09 invariants/java/daikon
```

Some warnings about analyzing historical versions of Daikon:

1. When analyzing 2002 (and at least parts of 2003) you should be careful to use the branch, not the trunk. Or, you could analyze both (but as a single development effort, not as separate efforts).
2. When a programmer periodically merged changes from the branch to the trunk (or vice versa), that operation resulted in very large checkins. The times at which these merges occurred is indicated in file `invariants/java/daikon/merging.txt` in the repository; for example, this happened 34 times during calendar year 2002. CVS checkins for the branch properly attribute and time-stamp the work that appears as a single large checkin on the trunk.
3. There may be long periods of time in the branch (respectively, the trunk) with no checkins, but that does not necessarily indicate a lacuna in development, as checkins might have occurred in the meanwhile in the trunk (respectively, the branch).

Appendix A File formats

This chapter contains information about the file format of Daikon's input files. It is of most information to those who wish to write a front end, also known as an instrumenter (see Section "Front ends and instrumentation" in *Daikon User Manual*). A new front end enables Daikon to detect invariants in another programming language.

Daikon's input is conventionally one or more `.dtrace` data trace files. (Another, optional type of input file for Daikon is a *splitter* info file; see Section "Splitter info file format" in *Daikon User Manual*.) A trace file is a text file that consists of newline-separated records. There are two basic types of records that can appear in Daikon's input: program point declarations, and trace records. The declarations describe the structure of the trace records. The trace records contain the data on which Daikon operates — the run-time values of variables in your program.

Each declaration names an instrumented program point and lists the variables at that program point. A program point is a location in the program, such as a specific line number, or a specific procedure's entry or exit. An instrumented program point is a place where the instrumenter may emit a trace record. A program point declaration may be repeated, so long as the declarations match exactly (any declarations after the first one have no effect).

A data trace record (also known as a *sample*) represents one execution of a program point. The record specifies the program point and gives the run-time values of each variable. The list of variables in the data trace record must be identical to that in the corresponding declaration. For a given program point, the declaration must precede the first data trace record for the program point. It is not required that all the program point declarations appear before any of the data trace records.

There exist some other declaration-related records; see Section A.3.1 [Declaration-related records], page 32.

A.1 Declarations in a separate file

Instead of placing both declarations and data trace records in a single file, it is permitted to place the declarations in one or more `.decls` *declaration files* while leaving the data trace records in the `.dtrace` file. This can be convenient for tools that perform a separate instrumentation step, such as `dfepl` (see Section "dfepl" in *Daikon User Manual*). Such a tool takes as input a target program to be analyzed, and produces two outputs: a `.decls` file and an instrumented program. Executing the instrumented program produces a `.dtrace` file containing data trace records for all the program points that appear in the `.decls` file. This approach works fine and is easier to implement in certain situations, but has a few disadvantages. It requires the user to perform at least two steps — instrumentation and execution — and the existence of two versions of the program (instrumented and uninstrumented) can lead to confusion or extra work. It is also more convenient to have a single file that contains all information about a program, rather than multiple `.decls` files that must be associated with the `.dtrace` file.

It is also permitted for a declaration to appear more than once in Daikon's input. The declaration must be identical in all occurrences. This is useful when running Daikon with multiple `.dtrace` files, each of which contains its own declarations.

A.2 File format conventions

Daikon files are textual, to permit easier viewing and editing by humans. Each record is separated by one or more blank lines. To permit easier parsing by programs, each piece of information in a record appears on a separate line.

Outside a record, any line starting with a pound sign (#) or double slashes (//) is ignored as a comment. Comments are not permitted inside a record.

A.3 Declarations

The trace file (or declaration file) first states the declaration file format version number (see Section A.3.1.1 [Declaration version], page 32). It may also specify some other information about the file (see Section A.3.1 [Declaration-related records], page 32). Then, it defines each program point and its variables.

Indentation is ignored, so it may be used to aid readability. Fields with defaults can be omitted.

As a rule, each line of the declaration file is of the form `<field-name> <field-value>`.

Some additional details about the declaration file format appear in file `daikon/doc/decl_format.txt` in the Daikon source code.

A.3.1 Declaration-related records

A.3.1.1 Declaration version

The declaration version record must be the first record in the file.

The declaration version record is as follows:

```
decl-version <version>
```

The current version is 2.0.

Previous versions (see Section A.6 [Version 1 Declarations], page 45) did not include a version field and are identified by the lack of this field.

A.3.1.2 Input-language declaration

You can specify the language in which the program was written with a record of the form

```
input-language <language>
```

The language string is arbitrary and is not used.

A.3.1.3 Variable comparability

The Variable comparability record indicates how the comparability field of a variable declaration should be interpreted.

Its format is:

```
var-comparability <comparability-type>
```

The possible values for *comparability-type* are `implicit` and `none`.

`implicit` means ordinary comparability as described in Section A.3.3 [Variable declarations], page 34. (The name `implicit` is retained for historical reasons.)

This record is optional. The `implicit` type is the default.

A.3.1.4 ListImplementors declaration

This declaration indicates classes that implement the `java.util.List` interface, and should be treated as sequences for the purposes of invariant detection. The syntax is as follows:

```
ListImplementors
<classname1>
<classname2>
...
```

Each `classname` is in Java format (for example, `java.util.LinkedList`).

The `--list_type` command-line option to Daikon can also be used to specify classes that implement lists; see Section "Options to control invariant detection" in *Daikon User Manual*.

A.3.2 Program point declarations

The format of a program point declaration is:

```
ppt <ppt-name>
<ppt-info>
<ppt-info>
...
<variable-declaration>
<variable-declaration>
...
```

The program point name can include any character. In the declaration file, blanks must be replaced by `\_`, and backslashes must be escaped as `\\`. Program point names must be distinct.

While Daikon does not infer program point relationships from `ppt-name`s, it does require these names to conform to a set syntax. The following patterns are for the `enter`, `subexit`, `class` and `object` `ppt-type`s, respectively.

```
<fully qualified class name>.<method/function name>(<argument types>):::ENTER
<fully qualified class name>.<method/function name>(<argument types>):::EXIT<id>
<fully qualified class name>:::CLASS
<fully qualified class name>:::OBJECT
```

Since in most languages a method or function may have multiple exit points, the `ppt-name` for a `subexit` `ppt-type` must be appended with a unique id. Typically, this is the corresponding source line number.

The following information about the program point (`ppt-info`) can be specified:

- `ppt-type <type>`

  Specifies the *type* of the program point. Possible program point types are `point`, `class`, `object`, `enter`, `exit`, `subexit`. Except for `point` all of these types are related to the program point hierarchy (see Section 4.2 [Dataflow hierarchy], page 12).

  A `point` program point is one that is *not* involved in a program point hierarchy. This is normally used when the input is not from a programming language or when there is no dataflow hierarchy.

- `flags <flags>`

  Specifies one or more flags for this `ppt`. The possible flags are: `static`, `enter`, `exit`, `private`, `return`. It should be noted that neither Daikon nor Kvasir currently use or output this field.

- `parent <relation-type> <parent-ppt-name> <relation-id>`

  Specifies the program point hierarchy (Section 4.2 [Dataflow hierarchy], page 12).

  In particular, each `parent` field names one parent of this program point. A parent program point is a point whose samples should include all of the samples at this program point. For example, an object program point is a parent of each of the method program points in that object.

  The *relation-type* is the type of parent-child relationship in the hierarchy. There are a few relationship types used internally by Daikon, but the only one output to the data files is `parent`. A `parent` relationship is one where the program points themselves are explicitly related, such as an enter and an exit point. All of the variables at one of the points exists at the other. A `user` relation is one where a class is used at another point, such as at an enter point. For example, if a reference to class A were passed to routine '`r1`', the values found at enter and exit of '`r1`' could be applied to the class/object program point for A. By default `user` relations are not used because they can be recursive.

  The *relation-id* is a unique integer that identifies this parent relation. They are used when defining the specific parent relations for variables.

Multiple parent fields can be specified.

A.3.3 Variable declarations

The format of a variable declaration is:

```
variable <name>
  <variable-info>
  <variable-info>
  ...
```

The variable name is arbitrary, but for clarity, it should match what is used in the programming language. All characters are legal in a name, but blanks must be represented as `\_` and backslashes as `\\`.

If the variable is an array, `..` marks the location of array indices within the variable name. Some examples of names are:

```
this.theArray
this.theArray[..]
this.stack.getClass()
```

The following information about the variable (`variable-info`) can be specified:

- `var-kind <kind> [<relative-name>]`

  Specifies the variable kind. Possible values are: `field`, `function`, `array`, `variable`, `return`. If `field` or `function` are specified, the relative name of the field or function must be specified. For example, if the variable is `this.theArray`, the relative name is `theArray`. Pointers to arrays are of type `field`. The arrays themselves (a sequence of values) are of type `array`. A var-kind entry is required in each variable block.

- `enclosing-var <enclosing-var-name>`

  The variable that contains this variable. Required for fields and arrays. Required for functions that represent an instance method. Forbidden for functions that specify a static method or a function in a non-object-oriented language. A variable is specified by its name. The named variable must be defined at the same program point. If a variable is omitted (e.g., by the omit-var switch), any variable for which it is the enclosing variable must be omitted as well.

  For example, if the variable is `this.theArray`, the enclosing variable is `this`.

- `reference-type pointer|offset`

  Specifies the kind of reference for variables which are structures or classes. The possible values are `pointer` or `offset`. In C, `pointer` is used if the variable is a pointer, `offset` is used when the structure is placed inline. Pointer would be used for all references to Java objects. Defaults to pointer.

- `array <dim>`

  The number of array dimensions inherited or declared by this variable. The valid values are 0 or 1. This should be specified for any variable that has multiple values. If not specified it defaults to 0. Future versions of Daikon may support more levels of arrays.

- `dec-type <language-declaration>`

  This is what the programmer used in the declaration of the variable. Names for standard types should use Java's names (e.g., `int`, `boolean`, `java.lang.String`, etc.), but names for user-defined or language-specific types can be arbitrary strings. A `dec-type` entry is required in each variable block.

- `rep-type <daikon-type>`

  This describes what will appear in the data trace file. For instance, the declared type might be `char[..]` but the representation type might be `java.lang.String`. Or, the declared type

might be `Object` but the representation type might be `hashcode`, if the address of the object is written to the data trace file. A rep-type entry is required in each variable block.

The representation type should be one of `boolean`, `int`, `hashcode`, `double`, or `java.lang.String`; or an array of one of those (indicated by a `[..]` suffix).

`hashcode` is intended for unique object identifiers like memory addresses (pointers) or the return value of Java's `Object.hashCode` method. `hashcode` is treated like `int`, except that the hashcode values are considered uninteresting for the purposes of output. For example, Daikon will print '*var* `has only one value`' instead of '*var* `== 0x38E8A`'.

- `flags <flags>`

  One or more flags may optionally be specified. Possible values are:

  - `is_param`

    Indicates that a given variable is a parameter to a procedure. Some procedures reassign parameters — essentially using them as local variables. Such uses are not relevant to the procedure's external specification. The `is_param` flag causes Daikon not to print certain invariants, if the variable has been reassigned.

    1. Invariants that use the parameter variable `p` in its post-state form are not printed.
    2. Invariants that use fields of `p` (such as `p.x`) are printed only if `p` has not changed.
    3. Some immutable characteristics, such as the size of arrays and data types, are not printed. (These can change only if `p` is changed, but then, `p` would no longer be interesting.)

  - `no_dups`

    Indicates that a collection can not contain duplicates. If it cannot, Daikon does not check for some invariants that only have meaning for collections that can contain duplicate elements.

  - `not_ordered`

    Indicates that the order of a collection does not have meaning. In this case, Daikon does not check for element-wise comparisons between it and other collections.

  - `nomod`

    Indicates that the variable can never be modified. For example, a variable declared `static final` in Java.

  - `synthetic`

    Indicates that the variable was added by the front end and is not manifest in the input program.

  - `classname`

    Indicates that the variable indicates the `classname` of its enclosing variable.

  - `to_string`

    Indicates that the variable is the string representation of its enclosing variable.

  - `non_null`

    Indicates that the variable can't take on a null value. In this case, Daikon will not check for the '`NonZero`' invariant.

  - `is_property`

    Indicates that the variable is a C# property (set by the Celeriac front end). It is used in output formatting to remove the parentheses from a method call.

  - `is_enum`

    Indicates that the variable is an enumerated type (set by the Celeriac front end). Daikon uses this information to suppress obvious invariants.

  - `is_readonly`

    Indicates that the variable is read only (set by the Celeriac front end). Daikon uses this information to suppress obvious invariants.

- `comparability <comparability-key>`

  The *comparability-key* indicates which other variables are comparable to this one. The information specified here might have been obtained dynamically, or via type-inference-based static analysis, or in some other manner.

  A comparability for a non-array type is a signed integer. Two variables at the same program point are considered comparable if both integers are the same, or if either integer is negative (that is, a negative number means "comparable to every other variable"). A comparability for an array type contains an integer for each index and for the contents; for instance, '`8[5]`' means that the array elements have comparability '`8`' and the array indices have comparability '`5`'. Similarly, '`5[22][17]`' is a comparability for a two-dimensional array. An array comparison succeeds if comparisons over each component succeed.

  Variables at different program points are never compared to one another. Use of the same number at different program points does not indicate any relationship between the variables, and a given variable may have a different comparability integer at different program points.

  As an example, in the following code:

```
int sum(int len, int[] a) {
  int sum=0;
  for (int i=0; i++; i<len)
    sum += a[i];
  return sum;
}
```

  variables `i` and `len` are comparable to one another (and to indices of array `a`). Furthermore, the result is comparable to the elements of array `a`. The comparability keys for these variables might look like

```
len     - comparability 5
a       - comparability 8[5]
return  - comparability 8
```

  A comparability entry is required in each variable block.

- `parent <parent-ppt> <relation-id> [<parent-variable>]`

  Optionally specifies the parent variable of this variable in the program point/variable hierarchy. The *parent-ppt* is the name of the parent program point. The *relation-id* must be one of the relationship ids specified for this program point. The *parent-variable* is the name of this variable's parent in the parent program point. If the names are the same, it can be omitted.

- `constant <value>`

  Optionally specifies a constant value for this variable. If the variable has a compile-time constant value that is specified in the declaration, then the variable must be omitted from the data trace records.

- `function-args <arg1> <arg2> ...`

  If this variable is computed as a function of some other variable, specifies the arguments to the function; otherwise, the `function-args` line must be omitted. The arguments are specified by their external variable name. Multiple arguments are blank separated. For example

```
function-args a.b this.f1
```

  specifies that the function takes two arguments which are `a.b` and `this.f1`. As with enclosing variables, each of the arguments must be defined as variables.

- `min-value <v>`

  Optionally specifies the minimum possible value for this variable (for instance, due to language-specific restrictions on the type). This enables Daikon to suppress invariants that would be "obvious" in that case, such as `var >= v`.
- `max-value <v>`

  Optionally specifies the maximum possible value for this variable (for instance, due to language-specific restrictions on the type). This enables Daikon to suppress invariants that would be "obvious" in that case, such as `var <= v`.
- `min-length <v>`

  Optionally specifies the minimum possible length for this variable (for instance, due to language-specific restrictions on the type). This enables Daikon to suppress invariants that would be "obvious" in that case, such as `size(var) >= v`.
- `max-length <v>`

  Optionally specifies the maximum possible length for this variable (for instance, due to language-specific restrictions on the type). This enables Daikon to suppress invariants that would be "obvious" in that case, such as `size(var) <= v`.
- `valid-values [<v1> <v2> ... <vN>]`

  Optionally specifies all the possible values for this variable (for instance, due to language-specific restrictions on the type). This enables Daikon to suppress invariants that would be "obvious" in that case, such as `v is one of {v1, v2, ..., vN}`.

A.4 Data trace records

A data trace record (also known as a *sample*) contains run-time value information. Its format is:

```
<program-point-name>
this_invocation_nonce
<nonce-string>
<varname-1>
<var-value-1>
<var-modified-1>
<varname2>
<var-value-2>
<var-modified-2>
...
```

In other words, the sample record contains:

- name of the program point
- optionally, an arbitrary string (a nonce) used to match up procedure entries (whose names conventionally end with `:::ENTER`) with procedure exits (whose names conventionally end with `:::EXIT`). This is necessary in concurrent systems because there may be several invocations of a procedure active at once and they do not necessarily follow a stack discipline, being exited in the reverse order of entry. For non-concurrent systems, this nonce is not necessary, and both the line `this_invocation_nonce` and the nonce value may be omitted.
- for each variable:
  - name
  - value
    - if an integer: sequence of digits, optionally preceded by a minus sign. Boolean values are written as the number 0 (for false) or the number 1 (for true). For pointers, the value may be `null`.

- if a string: characters surrounded by double-quotes. Internal double-quotes and backslashes are escaped by a backslash. Newlines and carriage returns are represented as '`\n`' and '`\r`', respectively.
- if an array: open bracket (`[`), elements separated by spaces, close bracket (`]`). (Also, the array name should end in '`[..]`'; use '`a[..]`' for array contents, but '`a`' for the identity of the array itself.)

The value representation may also be the string `nonsensical`; see Section A.4.1 [Nonsensical values], page 38. A string or array *value* is never `null`. A *reference* to a string or array may be `null`, in which case the string or array value is printed as `nonsensical`.

- modified? (0, 1, or 2). This value is 0 if the variable has not been assigned to since the last time this program point was executed, and 1 if the variable has been assigned to since then. It is safe for an implementation to always set it to 1. It is also safe to always set it to 0, because Daikon corrects obviously incorrect modification bits (such as 0 for a never-before-seen value).

  The special value 2 should be used only (and always) when the value field is `nonsensical`.

The variables should appear in the same order as they did in the declaration of the program point, without omissions or additions.

A.4.1 Nonsensical values for variables

Some trace variables and derived variables may not have a value because the expression that computes it cannot be evaluated. In such a circumstance, the value is said to be nonsensical, it is written in the trace file as `nonsensical`, and its modified field must be 2. Examples include

- `x` when `x` is uninitialized or deallocated,
- `x.y` when `x` is null (or uninitialized or deallocated)
- `a[i]` when `i` is outside the bounds of `a` (or uninitialized or deallocated, or `a` is null, uninitialized, or deallocated)

For trace variables, it is the responsibility of the front end to perform a check at run time whenever a variable's value is about to be output to the trace, and to output the value '`nonsensical`' (see Section A.4.1 [Nonsensical values], page 38) rather than crashing the program or outputting an uninitialized or meaningless value. (Determining when an expression's value is meaningless is the most challenging part of writing an instrumenter for a language like C, since it requires tracking memory allocation and initialization.) For derived variables created by Daikon, Daikon does the same thing, setting values to '`nonsensical`' when appropriate. For controlling Daikon's output in the presence of nonsensical values, see the `daikon.Daikon.guardNulls` configuration option (see Section "General configuration options" in *Daikon User Manual*).

A.4.2 Variables that do not appear in trace records

A trace record should contain exactly the same variables as in the corresponding declaration. There is one exception: for efficiency, compile-time constants (e.g., static final variables in Java) are omitted from the trace record, since they would have the same value every time.

Furthermore, neither the declarations nor the trace records contain derived variables (see Section "Variable names" in *Daikon User Manual*).

A.5 Example files

Here are portions of two files `StackArTester.decls` and `StackArTester.dtrace`, for a Java class that implements a stack of integers using an array as the underlying data structure. You can see many more examples by simply running an existing front end on some Java, C, or Perl programs and viewing the resulting files.

A.5.1 Example declaration file

This is part of the file `StackArTester.decls`, a declaration file for the `StackAr.java` program (see Section "StackAr example" in *Daikon User Manual*).

```
ppt DataStructures.StackAr.push(java.lang.Object):::ENTER
ppt-type enter
parent parent DataStructures.StackAr:::OBJECT 1
variable this
  var-kind variable
  dec-type DataStructures.StackAr
  rep-type hashcode
  flags is_param nomod
  comparability 22
  parent DataStructures.StackAr:::OBJECT 1
variable this.theArray
  var-kind field theArray
  enclosing-var this
  dec-type java.lang.Object[]
  rep-type hashcode
  flags nomod
  comparability 22
  parent DataStructures.StackAr:::OBJECT 1
variable this.theArray.getClass().getName()
  var-kind function getClass().getName()
  enclosing-var this.theArray
  dec-type java.lang.Class
  rep-type java.lang.String
  function-args this.theArray
  flags nomod synthetic classname
  comparability 22
  parent DataStructures.StackAr:::OBJECT 1
variable this.theArray[..]
  var-kind array
  enclosing-var this.theArray
  array 1
  dec-type java.lang.Object[]
  rep-type hashcode[]
  flags nomod
  comparability 22
  parent DataStructures.StackAr:::OBJECT 1
variable this.theArray[..].getClass().getName()
  var-kind function getClass().getName()
  enclosing-var this.theArray[..]
  array 1
  dec-type java.lang.Class[]
  rep-type java.lang.String[]
  function-args this.theArray[]
  flags nomod synthetic classname
  comparability 22
  parent DataStructures.StackAr:::OBJECT 1
variable this.topOfStack
  var-kind field topOfStack
  enclosing-var this
  dec-type int
  rep-type int
  flags nomod
  comparability 22
  parent DataStructures.StackAr:::OBJECT 1
```

```
variable DataStructures.StackAr.DEFAULT_CAPACITY
  var-kind variable
  dec-type int
  rep-type int
  constant 10
  flags nomod
  comparability 22
  parent DataStructures.StackAr:::OBJECT 1
variable x
  var-kind variable
  dec-type java.lang.Object
  rep-type hashcode
  flags is_param nomod
  comparability 22
variable x.getClass().getName()
  var-kind function getClass().getName()
  enclosing-var x
  dec-type java.lang.Class
  rep-type java.lang.String
  function-args x
  flags nomod synthetic classname
  comparability 22

ppt DataStructures.StackAr.push(java.lang.Object):::EXIT103
ppt-type subexit
parent parent DataStructures.StackAr:::OBJECT 1
variable this
  var-kind variable
  dec-type DataStructures.StackAr
  rep-type hashcode
  flags is_param nomod
  comparability 22
  parent DataStructures.StackAr:::OBJECT 1
variable this.theArray
  var-kind field theArray
  enclosing-var this
  dec-type java.lang.Object[]
  rep-type hashcode
  flags nomod
  comparability 22
  parent DataStructures.StackAr:::OBJECT 1
variable this.theArray.getClass().getName()
  var-kind function getClass().getName()
  enclosing-var this.theArray
  dec-type java.lang.Class
  rep-type java.lang.String
  function-args this.theArray
  flags nomod synthetic classname
  comparability 22
  parent DataStructures.StackAr:::OBJECT 1
variable this.theArray[..]
  var-kind array
  enclosing-var this.theArray
  array 1
  dec-type java.lang.Object[]
  rep-type hashcode[]
  flags nomod
  comparability 22
```

```
    parent DataStructures.StackAr:::OBJECT 1
  variable this.theArray[..].getClass().getName()
    var-kind function getClass().getName()
    enclosing-var this.theArray[..]
    array 1
    dec-type java.lang.Class[]
    rep-type java.lang.String[]
    function-args this.theArray[]
    flags nomod synthetic classname
    comparability 22
    parent DataStructures.StackAr:::OBJECT 1
  variable this.topOfStack
    var-kind field topOfStack
    enclosing-var this
    dec-type int
    rep-type int
    flags nomod
    comparability 22
    parent DataStructures.StackAr:::OBJECT 1
  variable DataStructures.StackAr.DEFAULT_CAPACITY
    var-kind variable
    dec-type int
    rep-type int
    constant 10
    flags nomod
    comparability 22
    parent DataStructures.StackAr:::OBJECT 1
  variable x
    var-kind variable
    dec-type java.lang.Object
    rep-type hashcode
    flags is_param nomod
    comparability 22
  variable x.getClass().getName()
    var-kind function getClass().getName()
    enclosing-var x
    dec-type java.lang.Class
    rep-type java.lang.String
    function-args x
    flags nomod synthetic classname
    comparability 22

  ppt DataStructures.StackAr:::CLASS
  ppt-type class
  variable DataStructures.StackAr.DEFAULT_CAPACITY
    var-kind variable
    dec-type int
    rep-type int
    constant 10
    flags nomod
    comparability 22

  ppt DataStructures.StackAr:::OBJECT
  ppt-type object
  parent parent DataStructures.StackAr:::CLASS 1
  variable this
    var-kind variable
    dec-type DataStructures.StackAr
```

```
    rep-type hashcode
    flags is_param nomod
    comparability 22
  variable this.theArray
    var-kind field theArray
    enclosing-var this
    dec-type java.lang.Object[]
    rep-type hashcode
    flags nomod
    comparability 22
  variable this.theArray.getClass().getName()
    var-kind function getClass().getName()
    enclosing-var this.theArray
    dec-type java.lang.Class
    rep-type java.lang.String
    function-args this.theArray
    flags nomod synthetic classname
    comparability 22
  variable this.theArray[..]
    var-kind array
    enclosing-var this.theArray
    array 1
    dec-type java.lang.Object[]
    rep-type hashcode[]
    flags nomod
    comparability 22
  variable this.theArray[..].getClass().getName()
    var-kind function getClass().getName()
    enclosing-var this.theArray[..]
    array 1
    dec-type java.lang.Class[]
    rep-type java.lang.String[]
    function-args this.theArray[]
    flags nomod synthetic classname
    comparability 22
  variable this.topOfStack
    var-kind field topOfStack
    enclosing-var this
    dec-type int
    rep-type int
    flags nomod
    comparability 22
  variable DataStructures.StackAr.DEFAULT_CAPACITY
    var-kind variable
    dec-type int
    rep-type int
    constant 10
    flags nomod
    comparability 22
    parent DataStructures.StackAr:::CLASS 1
```

A.5.2 Example data trace file

This is part of file `StackArTester.dtrace`, which you can create by running the instrumented `StackAr.java` program (see Section "StackAr example" in *Daikon User Manual*). This excerpt contains only the first two calls to `push` and the first return from `push`, along with the associated object program point records; omitted records are indicated by ellipses.

```
...

DataStructures.StackAr.push(java.lang.Object):::ENTER
this_invocation_nonce
104
this
812272602
1
this.theArray
312077835
1
this.theArray.getClass().getName()
"java.lang.Object[]"
1
this.theArray[..]
[null]
1
this.theArray[..].getClass().getName()
[null]
1
this.topOfStack
-1
1
x
1367164551
1
x.getClass().getName()
"DataStructures.MyInteger"
1

...

DataStructures.StackAr.push(java.lang.Object):::EXIT103
this_invocation_nonce
104
this
812272602
1
this.theArray
312077835
1
this.theArray.getClass().getName()
"java.lang.Object[]"
1
this.theArray[..]
[1367164551]
1
this.theArray[..].getClass().getName()
["DataStructures.MyInteger"]
1
this.topOfStack
0
1
x
1367164551
1
x.getClass().getName()
"DataStructures.MyInteger"
```

```
1

...

DataStructures.StackAr.push(java.lang.Object):::ENTER
this_invocation_nonce
159
this
2007069404
1
this.theArray
142345952
1
this.theArray.getClass().getName()
"java.lang.Object[]"
1
this.theArray[..]
[null]
1
this.theArray[..].getClass().getName()
[null]
1
this.topOfStack
-1
1
x
111632506
1
x.getClass().getName()
"DataStructures.MyInteger"
1

...

DataStructures.StackAr.push(java.lang.Object):::EXIT103
this_invocation_nonce
159
this
2007069404
1
this.theArray
142345952
1
this.theArray.getClass().getName()
"java.lang.Object[]"
1
this.theArray[..]
[111632506]
1
this.theArray[..].getClass().getName()
["DataStructures.MyInteger"]
1
this.topOfStack
0
1
x
111632506
1
```

```
x.getClass().getName()
"DataStructures.MyInteger"
1

...
```

A.6 Version 1 Declarations

This section describes the original version (1.0) of declaration records. These are now obsolete and should not be used.

A declarations file can contain program point declarations, `VarComparability` declarations, and `ListImplementors declarations`.

A.6.1 V1 Program point declarations

The format of a program point declaration is:

```
DECLARE
program-point-name
varname1
declared-type1 [# auxiliary-information1]
representation-type1 [= constant-value1]
comparable1
varname2
declared-type2 [# auxiliary-information2]
representation-type2 [= constant-value2]
comparable2
...
```

Program point information includes:

- name (*tag*) of this program point, an arbitrary string containing no tab or newline characters. This name contains information such as the class name or method name; what information is contained depends on which instrumenter is being used. See Section A.6.2 [V1 pptname format], page 47, for a full specification of the naming format.
- for each variable:
  - name: a string containing no tabs or newlines. See Section "Variable names" in *Daikon User Manual*.
  - declared type: this is what the programmer used in the declaration of the variable. Array types must be suffixed by the proper number of '`[]`' to indicate their dimensionality. Names for standard types should use Java's names (e.g., `int`, `boolean`, etc.), but names for user-defined or language-specific types can be arbitrary strings.
  - auxiliary information: optionally, Daikon can be given information about the meaning of the variable to help it better interpret the values it later sees. Information is provided as a comma-separated list of items, with each item in the form of '`key = value`'. Unrecognized keys are silently ignored. All values are either '`true`' or '`false`'. Mainly, this information is used for collections, which are presented to Daikon as arrays. Valid keys are:

    `hasDuplicates`
    : Whether a collection can contain duplicates. If it cannot, Daikon does not check for some invariants that only have meaning for collections that can contain duplicate elements.

`hasOrder` Whether order has meaning for a collection. If order does not have meaning in a collection, then Daikon does not check for element-wise comparisons between it and other collections.

`hasNull` Whether zero has the special meaning null for the variable or collection. If it does, then Daikon checks for whether a value or the elements in a collection are null.

`nullTerminated`
Whether a collection has a value (usually null) that ends its representation. If it does, then Daikon looks at the collection's size and at the collection's size-1 as "interesting" values. If it does not, then Daikon only looks at the collection's size.

`isParam` Whether a given variable is a parameter to a method. If a variable is a parameter, Daikon avoids printing some information that would be considered uninteresting for parameters. First, invariants that use the parameter variable `p` in its post-state form are not printed. Second, invariants that use fields of `p` (such as `p.x`) are printed only if `p` has not changed. Lastly, some immutable characteristics, such as the size of arrays and data types are not printed (both can be changed if `p` is changed, but then, `p` would no longer be interesting).

- representation type: this describes what will appear in the data trace file. For instance, the declared type might be `char[]` but the representation type might be `java.lang.String`. Or, the declared type might be `Object` but the representation type might be `hashcode`, if the address of the object is written to the data trace file.

  The representation type should be one of `boolean`, `int`, `hashcode`, `double`, or `java.lang.String`; or an array of one of those (indicated by a `[]` suffix, as in Java). Hashcodes are treated like integers, except that their actual values are considered uninteresting for the purposes of output; they are intended for unique object identifiers like memory addresses or the return value of Java's `Object.hashCode` method.

  The representation type may optionally be followed by an equals sign and a value; in that case, the variable is known to have a compile-time constant value and should be omitted from the data trace file.

- comparable variables. This information indicates which other variables are comparable to this one.

  The point of comparability is that Daikon should not compare unrelated quantities. For example, each person's height in centimeters may always be less than their birth year, but it is not helpful for Daikon to output '`height < birthyear`', because the two variables are measuring incomparable quantities. (In this case, the variables use different units of measurement.)

  Variable comparability information helps Daikon to avoid computing information over unrelated variables. This saves time and (more importantly) improves the quality of Daikon's output. For more details, see the paper "Quickly detecting relevant program invariants".

  Variable comparability information may be obtained dynamically (see Section "Dynamic abstract type inference (DynComp)" in *Daikon User Manual*), via type-inference based analysis, or in some other manner. In any event, Daikon reads it from the variable declarations.

  A comparability for a non-array type is a signed integer. Two variables at the same program point are considered comparable if both integers are the same, *or* if either integer is negative. A comparability for an array type must contain an integer for each index and for the contents; for instance, '`5[22][17]`' for a two-dimensional array. Comparisons succeed if comparisons over each component succeed.

  Regardless of comparability, variables at different program points are never compared to one another. Use of the same comparability integer at different program points does not indicate

any relationship between the variables, and a given variable may have a different comparability integer at different program points.

As an example, in the following code:

```
int sum(int len, int[] a) {
  int sum=0;
  for (int i=0; i++; i<len)
    sum += a[i];
  return sum;
}
```

variables `i` and `len` are comparable to one another (and to indices of array `a`). Furthermore, the result is comparable to the elements of array `a`. A declaration file for these variables might look like

```
len
int
int
5
a
int[]
int[]
8[5]
return
int
int
8
```

A.6.2 Program point name format specification

Instrumenting code creates a `.decls` file that contains program point names such as:

```
DataStructures.StackAr.push(java.lang.Object):::ENTER
DataStructures.StackAr.push(java.lang.Object):::EXIT99
PolyCalc.RatNum.RatNum(int, int):::ENTER
PolyCalc.RatNum.RatNum(int, int):::EXIT55
PolyCalc.RatNum.RatNum(int, int):::EXIT67
```

This section describes the format of these program point names. Someone writing an instrumenter for a new language must be sure to follow this format specification.

A program point name is a string with no tabs or newlines in it. The basic format is '*topLevel*`.`*bottomLevel*`:::`*pptInfo*'. For the first example given above, the top level of the hierarchy would be `DataStructures.StackAr`, the bottom level would be `push(java.lang.Object)`, and the program point information would be `ENTER`.

topLevel may contain any number of periods ('`.`'). *bottomLevel* and *pptInfo* may not contain any periods. The string '`:::`' may only appear once.

topLevel and *pptInfo* are required (i.e., they must be non-empty), as are the period to the right of *topLevel* and the colons to the left of *pptInfo*. However, *bottomLevel* is optional.

By convention, for Java *topLevel* consists of the class name, and *bottomLevel* consists of the method name and method signature.

For C, *topLevel* consists of a filename (or a single period for global functions), and *bottomLevel* could consist of a function name and signature. More precisely, names of C program points follow these conventions:

- Names of program points for file-static functions are prefixed with the name of the source file, with '`.`' characters mapped to '`_`', followed by a '`.`'.

- Names of program points for file-scope functions with external linkage are prefixed with '`..`'. For example, a global function program point might be named '`..main():::ENTER`'., the first period denoting that it is global in scope and the second denoting the separator between the *toplevel* and *bottomlevel* parts of the name.
- Names of C++ functions that are class or namespace members are prefixed with the name(s) of their classes or namespaces, with the C++ '`::`' syntax mapped onto the Java '`.`' syntax used by Daikon.

For an Input Output Automaton, *topLevel* consists of an Automaton name and *bottomLevel* consists of information for a transition state.

By convention, the entry and exit points for a function have names of a special form so that they can be associated with one another. (Currently, those names end with `:::ENTER` and `:::EXIT`.) This convention permits Daikon to generate pre-state variables (see Section "Variable names" in *Daikon User Manual*) automatically at procedure exit points, so front ends need not output them explicitly. When there are multiple exit points, then each one should be suffixed by a number (such as a line number, for example, `foo::EXIT22`). Daikon produces the main (non-numbered) `:::EXIT` point automatically. All the numbered exits should contain the same set of variables; in general, this means that local variables are not included at exit points. Daikon requires that declarations for `:::ENTER` program points appear before any declarations for matching `:::EXIT` program points.

Another convention is to have another program point whose *bottomLevel* is empty and whose *pptInfo* is `OBJECT`: for example, `StackAr:::OBJECT`. This contains the representation invariant (sometimes called the object invariant) of a class. This program point is created automatically by Daikon; it need not appear in a trace file.

A.6.3 V1 VarComparability declaration

There is a special `VarComparability` declaration that controls how the comparability field in program point declarations is interpreted. The default `VarComparability` is `implicit`, which means ordinary comparability as described in Section A.3.2 [Program point declarations], page 33. (The name `implicit` is retained for historical reasons.) You can override it as follows:

```
VarComparability
none
```

As with all records in Daikon input files, a blank line is required between this record and the next one.

A.6.4 V1 ListImplementors declaration

This declaration indicates classes that implement the `java.util.List` interface, and should be treated as sequences for the purposes of invariant detection. The syntax is as follows:

```
ListImplementors
<classname1>
<classname2>
...
```

Each `classname` is in Java format (for example, `java.util.LinkedList`).

The `--list_type` command-line option to Daikon can also be used to specify classes that implement lists; see Section "Options to control invariant detection" in *Daikon User Manual*.

General Index

R

S

T

U

V