Daikon Invariant Detector User Manual

Daikon version 4.6.3

December 18, 2009

Table of Contents

1 Introduction

Daikon is an implementation of dynamic detection of likely invariants; that is, the Daikon invariant detector reports likely program invariants. An invariant is a property that holds at a certain point or points in a program; these are often seen in assert statements, documentation, and formal specifications. Invariants can be useful in program understanding and a host of other applications. Examples include ".field > abs(y)"; "y = 2*x+3"; "array a is sorted"; "for all list objects lst, lst.next.prev = lst"; "for all treenode objects n, n.left.value < n.right.value"; "p != null => p.content in myArray"; and many more. You can extend Daikon to add new properties (see Chapter 6 [Enhancing Daikon output], page 52).

Dynamic invariant detection runs a program, observes the values that the program computes, and then reports properties that were true over the observed executions. Daikon can detect properties in C, C++, Eiffel, Java, and Perl programs; in spreadsheet files; and in other data sources. (Dynamic invariant detection is a machine learning technique that can be applied to arbitrary data.) It is easy to extend Daikon to other applications; as one example, an interface exists to the Java PathFinder model checker.

Daikon is freely available for download from `http://pag.csail.mit.edu/daikon/download/`. The distribution includes both source code and documentation, and Daikon's license permits unrestricted use (see Section 10.2 [License], page 166). Many researchers and practitioners have used Daikon; those uses, and Daikon itself, are described in various publications.

More information on Daikon can be found in the Daikon Developer Manual (see section "Top" in *Daikon Developer Manual*). For instance, the Daikon Developer Manual indicates how to extend Daikon with new invariants, new derived variables, and front ends for new languages. It also contains information about the implementation and about debugging flags.

1.1 Mailing lists

The following mailing lists (and their archives) are available:

'`daikon-announce@lists.csail.mit.edu`'
: A low-volume, announcement-only list. For example, announcements of new releases are sent to this list. To subscribe, visit `https://lists.csail.mit.edu/mailman/listinfo/daikon-announce`.

'`daikon-discuss@lists.csail.mit.edu`'
: A moderated list for the community of Daikon users. Use it to share tips and successes, and to get help with questions or problems (after checking the documentation). To subscribe, visit `https://lists.csail.mit.edu/mailman/listinfo/daikon-discuss`.

'`daikon-developers@lists.csail.mit.edu`'
: This list goes to the Daikon maintainers. Use it for bug reports, suggestions, and the like. If you are an active contributor to Daikon, you may send mail to the list asking to be added.

2 Installing Daikon

Shortcut for the impatient: skip directly to installation instructions (see Section 2.2.2 [Unix/Linux/MacOS installation], page 3) (see Section 2.2.3 [Windows installation], page 4).

There are two main ways to install Daikon. See Section 2.1 [Simple installation], page 2, for a simple 2-step installation process that is adequate for the needs of many users. See Section 2.2 [Complete installation], page 2, for a complete installation, in 4 easy steps, that permits you to use all of the functionality in the Daikon distribution.

Use the simple installation instructions if you only wish to detect invariants in Java programs. Use the complete installation instructions (which are easy to follow) if you wish to detect invariants in C or Perl programs, if you wish to run certain other programs distributed with Daikon in its '`bin/`' directory, or if you wish to edit and recompile the source code of Daikon itself.

Differences from previous versions of Daikon appear in the file '`doc/CHANGES`' in the distribution. To be notified of new releases, or to join discussions about Daikon, subscribe to one of the mailing lists (see Section 1.1 [Mailing lists], page 1).

2.1 Simple installation instructions

Daikon is written in Java. In order to run Daikon, all you really need is the '`daikon.jar`' file, which is included in the distribution or can be downloaded separately from `http://pag.csail.mit.edu/daikon/download/daikon.jar`. Place '`daikon.jar`' on your classpath so that Java can find it. You are now ready to use Daikon!

There are two additional requirements. You must have a Java 5.0 (or later) JVM (Java Virtual Machine). The '`tools.jar`' file that comes with your JVM must also be on your classpath.

See Chapter 2 [Installing Daikon], page 2, for situations where you should follow the complete installation instructions of Section 2.2 [Complete installation], page 2. (Also, if you do not know how to add a jar file to your classpath, then use the complete installation instructions, which walk you through the process.)

2.2 Complete installation instructions

This section gives step-by-step instructions for installing Daikon.

Here is an overview of the steps. Details appear below; select the instructions for your operating system.

1. Download Daikon.
2. Place three commands in your shell initialization file.
3. Optionally, customize your installation.
4. Optionally, compile Daikon and build other tools by running `make`.

For more complete information on compiling Daikon, see section "Compiling Daikon" in *Daikon Developer Manual* in the Daikon Developer Manual.

2.2.1 Requirements for running Daikon

In order to run Daikon, you must have a Java 5.0 (or later) JVM (Java Virtual Machine). You must also have a Java 5.0 (or later) compiler.

2.2.1.1 Optional requirements for running Daikon

All the remaining requirements listed here are optional (they enable you to perform certain additional tasks with Daikon).

If you plan to use one of Daikon's source-based front ends, such as Mangel-Wurzel (see Section 7.4 [Mangel-Wurzel], page 122), then you need a compiler for whatever language your target programs are written in. For instance, if you wish to analyze C or C++ programs, you need a C or C++ compiler such as GCC. Source code and a compiler are not necessary if you plan to use one of Daikon's front ends that work on binaries, such as Chicory (see Section 7.1 [Chicory], page 88) and Kvasir (see Section 7.3 [Kvasir], page 100).

If you wish to edit the Daikon source code and re-compile Daikon, see section "Compiling Daikon" in *Daikon Developer Manual* in the Daikon developers manual.

2.2.2 Unix/Linux/MacOS installation

1. Choose the directory where you want to install Daikon; we'll call this the *daikonparent* directory. In this directory, download and unpack Daikon.

   ```
   cd daikonparent
   wget http://pag.csail.mit.edu/daikon/download/daikon.tar.gz
   tar zxf daikon.tar.gz
   ```

   This creates a '*daikonparent*/daikon/' subdirectory.
2. Place three commands in your shell initialization file: set two environment variables and source a Daikon startup file.

   sh/bash: If you use the sh or bash shell or their variants, add these three commands to your '.bashrc' or '.bash_profile' file:

   ```
   # The full pathname of the directory that contains Daikon
   export DAIKONDIR=$HOME/daikon
   # The full pathname of the directory that contains the Java JDK
   export JDKDIR=/usr/lib/j2sdk1.5-sun
   source $DAIKONDIR/bin/daikon.bashrc
   ```

   csh/tcsh: If you use the csh or tcsh shell or their variants, add these three commands to your '.cshrc' file:

   ```
   # The full pathname of the directory that contains Daikon
   setenv DAIKONDIR $HOME/daikon
   # The full pathname of the directory that contains the Java JDK
   setenv JDKDIR /usr/lib/j2sdk1.5-sun
   source $DAIKONDIR/bin/daikon.cshrc
   ```

   After editing your shell initialization file, either execute the commands you placed in it, or else log out and log back in to achieve the same effect.
3. Optionally, customize other variables. The customizable variables are listed in the Daikon startup file: 'daikon.bashrc' or 'daikon.cshrc'.

   You may customize them by setting environment variables, or by adding a Makefile.user file to directory $DAIKONDIR/java (it is automatically read at the beginning of the main Makefile, and prevents you from having to edit the main Makefile directly).

4. Compile Daikon and build other tools by running `make`. Note that this step is *not* required if you only want to use Daikon with its Java front end (Chicory).

   ```
   cd $DAIKONDIR
   make
   ```

   This builds the various executables used by Daikon, such as the C/C++ front end Kvasir (see Section 7.3.8 [Installing Kvasir], page 121) and the JDK for use with DynComp (see Section 7.2.1 [Instrumenting the JDK with DynComp], page 97). For more information about compiling Daikon, see section "Compiling Daikon" in *Daikon Developer Manual* in the Daikon Developer Manual.

   Note that running this make command may take 20 minutes or more, depending on your computer.

   Optionally, download other executables, such as the Simplify theorem prover (`http://www.hpl.hp.com/downloads/crl/jtk/`).

2.2.3 Windows installation

To perform a complete install on Windows, it is necessary to install the Cygwin toolset. After you have installed Daikon, you can run it using either Cygwin or the regular Windows shell (see Section 2.2.4 [Running Daikon under Windows], page 5).

The Cygwin toolset (available at `http://sources.redhat.com/cygwin/`) contains everything you need to compile and run Unix programs under Windows. You can install Cygwin by simply running the program found at `http://sources.redhat.com/cygwin/setup.exe`. The default installation of Cygwin is sufficient for installing Daikon.

1. Choose the directory where you want to install Daikon; we'll call this the *daikonparent* directory. In this directory, download and unpack Daikon.

   ```
   cd daikonparent
   wget http://pag.csail.mit.edu/daikon/download/daikon.tar.gz
   tar zxf daikon.tar.gz
   ```

   This creates a '*daikonparent*`/daikon/`' subdirectory.

2. Place three commands in your shell initialization file '`.bashrc`': set two environment variables and source a Daikon startup file. Do not use a Windows shell; use the Cygwin bash shell instead.

   ```
   # The full pathname of the directory that contains Daikon
   export DAIKONDIR=$HOME/daikon
   # The full Unix pathname of the directory that contains the Java JDK
   export JDKDIR=/cygdrive/c/Program Files/Java/jdk1.5.0_05
   source $DAIKONDIR/bin/daikon.bashrc
   ```

   Use the Cygwin/Unix path style (e.g., `/cygdrive/c/daikon`) rather than the windows path style (`C:\daikon`). Some users have reported problems when using pathnames with spaces. You can avoid the problem by using the `ln` command to add a symbolic link without spaces to `Program Files`.

   ```
   cd /cygdrive/c
   ln -s "Program Files" program_files
   ```

   and then the JDKDIR line becomes:

```
export JDKDIR=/cygdrive/c/program_files/Java/jdk1.5.0_05
```

After editing your shell initialization file, either execute the commands you placed in it, or else log out and log back in to achieve the same effect.

3. Optionally, customize other variables. The customizable variables are listed in the Daikon startup file: '`daikon.bashrc`' or '`daikon.cshrc`'.

   You may customize them by setting environment variables, or by adding a `Makefile.user` file to directory `$DAIKONDIR/java` (it is automatically read at the beginning of the main Makefile, and prevents you from having to edit the main Makefile directly).

   The one variable you must customize is to define the `OSTYPE` variable to be `cygwin`.

   ```
   OSTYPE = cygwin
   ```

4. Compile Daikon and build other tools by running `make`. Note that this step is *not* required if you only want to use Daikon with its Java front end (Chicory).

   ```
   cd $DAIKONDIR
   make
   ```

   This builds the various executables used by Daikon, such as the C/C++ front end Kvasir (see Section 7.3.8 [Installing Kvasir], page 121) and the JDK for use with DynComp (see Section 7.2.1 [Instrumenting the JDK with DynComp], page 97). For more information about compiling Daikon, see section "Compiling Daikon" in *Daikon Developer Manual* in the Daikon Developer Manual.

   Note that running this make command may take 20 minutes or more, depending on your computer. On Windows, running `make` requires that Cygwin (`http://www.cygwin.com/`) be installed.

   Optionally, download other executables, such as the Simplify theorem prover (`http://www.hpl.hp.com/downloads/crl/jtk/`), or the Z3 theorem prover (`http://research.microsoft.com/en-us/um/redmond/projects/z3/`; Z3 can replace Simplify but is only distributed for Windows).

2.2.4 Running Daikon under Windows

After you have installed Daikon under Windows (see Section 2.2.3 [Windows installation], page 4), you can run it either using native Windows utilities, or using the Cygwin environment — it's your choice.

Daikon is a **command-line application** (and so are its related programs, such as Chicory). You should invoke them from a command shell — either a Windows command shell or a Cygwin command shell — rather than by double-clicking their icons. In any event, double-clicking would not supply the proper arguments to the program.

2.2.4.1 Windows command line

The first option is to run Daikon using native Windows utilities. The is done in the normal fashion. The `CLASSPATH` must include either `$DAIKONDIR/daikon.jar` or `$DAIKONDIR/java` (if you have recompiled the Daikon source). The `CLASSPATH` should be specified in Windows format (Windows paths and semicolon separators).

2.2.4.2 Cygwin shell

The second option for Windows is to run Daikon using the Cygwin toolset (available at `http://sources.redhat.com/cygwin/`), which contains everything you need to compile and run Unix programs under Windows. You can install Cygwin by simply running the program found at `http://sources.redhat.com/cygwin/setup.exe`.

There is an incompatibility between Cygwin and programs compiled for Windows (such as Java). (The incompatibility does not exist if the program was compiled for Cygwin.) The incompatibility is that Windows programs use the semicolon ('`;`') as their path separator (for instance, for `CLASSPATH`), but Unix and Cygwin programs use the colon ('`:`') as their path separator.

The `Makefiles` used to build Daikon will change the Unix style `CLASSPATH` (e.g., .;/cygdrive/c/daikon/java) into a Windows style `CLASSPATH` (.;C:\daikon\java) when necessary. The `CLASSPATH` must be in Unix format for the build to work correctly.

However, when running any Java program (such as Daikon or Chicory), the `CLASSPATH` should be specified in Windows format. That is why the build instructions suggest specifying the `CLASSPATH` in the `$DAIKONDIR/java/Makefile.user` file — so that the required Unix definition is not part of the environment.

3 Example usage for Java, C/C++, Perl, and Eiffel

Detecting invariants involves two steps:

1. Obtain one or more data trace files by running your program under the control of a front end (also known as an instrumenter or tracer) that records information about variable values. You can run your program over one or more inputs of your own choosing, such as regression tests or a typical user input session. You may choose to obtain trace data for only part of your program; this can avoid inundating you with output, and can also improve performance.
2. Run the Daikon invariant detector over the data trace files (see Chapter 4 [Running Daikon], page 17). This detects invariants in the recorded information. You can view the invariants textually, or process them with a variety of tools.

Often, you can run a single command that performs both steps. Among other benefits, this can avoid the need to create the data trace file by sending trace information directly from the instrumented program to Daikon, which is called "online execution" of Daikon.

This section briefly describes how to obtain data traces for Java, C, Perl, and Eiffel programs, and how to run Daikon. For detailed information about these and other front ends that are available for Daikon, Chapter 7 [Front ends (instrumentation)], page 88.

3.1 Detecting invariants in Java programs

In order to detect invariants in a Java program, run the program using the Chicory front end (see Section 7.1 [Chicory], page 88) to create a data trace file, then run Daikon itself to detect invariants. With the '`--daikon`' option to Chicory, a single command performs both steps.

For example, if you usually run

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

then instead you would run

```
java daikon.Chicory --daikon mypackage.MyClass arg1 arg2 arg3
```

and the Daikon output is written to the terminal.

3.1.1 StackAr example

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

To detect invariants in the StackAr sample program, perform the following steps.

1. Compile the program with the '`-g`' switch to enable debugging symbols. (The program and test suite appear in the '`DataStructures`' subdirectory directory.)

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

2. Run the program under the control of the Chicory front end, pass the information to Daikon, print the inferred invariants, and write a binary representation of the invariants to '`StackArTester.inv.gz`'.

```
java daikon.Chicory --daikon DataStructures.StackArTester
```

Alternately, replacing the '`--daikon`' argument by '`--daikon-online`' has the same effect, but does not write a data trace file to disk.

If you wish to have more control over the invariant detection process, you can split the second step above into multiple steps. Then, the whole process would be as follows:

1. Compile the program with the '`-g`' switch to enable debugging symbols. (The program and test suite appear in the '`DataStructures`' subdirectory directory.)

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

2. Run the program under the control of the Chicory front end, in order to create a trace file named '`StackArTester.dtrace.gz`'.

   ```
   java daikon.Chicory DataStructures.StackArTester
   ```

3. Run Daikon on the trace file.

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

   Daikon can analyze multiple runs (executions) of the program. You can supply Daikon with multiple trace files:

   ```
   java daikon.Chicory --dtrace-file=StackArTester1.dtrace.gz DataStructures.StackArTe
   java daikon.Chicory --dtrace-file=StackArTester2.dtrace.gz DataStructures.StackArTe
   java daikon.Chicory --dtrace-file=StackArTester3.dtrace.gz DataStructures.StackArTe
   java daikon.Daikon StackArTester*.dtrace.gz
   ```

   (In this example, all the runs are identical, so multiple runs yield the same invariants as one run.)

4. Examine the invariants. (They were also printed to standard out by the previous step.) There are various ways to do this.

   - Examine the output from running Daikon. (You may find it convenient to capture the output in a file; add '`> StackAr.txt`' to the end of the command that runs Daikon.)
   - Use the PrintInvariants program to display the invariants.

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

     For more options to the PrintInvariants program, see Section 8.1.1 [Printing invariants], page 142.
   - Use the Annotate program to insert the invariants as comments into the Java source program.

     ```
     cd ..
     java daikon.tools.jtb.Annotate StackArTester.inv.gz \
          DataStructures/StackAr.java
     ```

     Now examine file '`DataStructures/StackAr.java-escannotated`'. For more information about the Annotate program, see Section 8.1.4 [Annotate], page 145.

3.1.2 Using DynComp with Java programs

DynComp can help to filter Daikon's output by omitting invariants involving unrelated variables (see Section 7.2 [DynComp for Java], page 94). To do so, run DynComp on the target program first, then pass the resulting '.decls' file to Chicory. The process would be as follows:

1. Compile the program with the '-g' switch to enable debugging symbols. (The program and test suite appear in the 'DataStructures' subdirectory of the 'StackAr' directory.)

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

2. Run the program with DynComp to generate comparability information. You should produce an instrumented version of the JDK first (see Section 7.2.1 [Instrumenting the JDK with DynComp], page 97). By default, comparability information is written to 'StackArTester.decls-DynComp'.

```
java daikon.DynComp DataStructures.StackArTester
```

3. Run the program under the control of the Chicory front end, including comparability information. Pass the information to Daikon, print the inferred invariants, and write a binary representation of the invariants to 'StackArTester.inv.gz'. Note that this runs the target program for a second time.

```
java daikon.Chicory --daikon \
     --comparability-file=StackArTester.decls-DynComp \
     DataStructures.StackArTester
```

You could split the third step into multiple steps, as described in Section 3.1.1 [StackAr example], page 7, to gain more control over the invariant detection process.

3.1.3 Understanding the invariants

This section examines some of the invariants for the StackAr example. For more help interpreting invariants, see Section 5.4 [Interpreting output], page 27.

The StackAr example is an array-based stack implementation. Take a look at 'DataStructures/StackAr.java' to get a sense of the implementation. Now, look at the first section of Daikon output.

```
======================================================================
StackAr:::OBJECT
this.theArray != null
this.theArray.getClass() == java.lang.Object[].class
this.topOfStack >= -1
this.theArray[this.topOfStack+1..] elements == null
this.theArray[0..this.topOfStack] elements != null
this.topOfStack <= size(this.theArray[])-1
======================================================================
```

These six annotations describe the representation invariant. The array is never null, and its runtime type is `Object[]`. The `topOfStack` index is at least -1 and is less than the length of the array. Finally, the elements of the array are non-null if their index is no more than `topOfStack` and are null otherwise.

Next, look at the invariants for the `top()` method. `top()` has two different exit points, at lines 78 and 79 in the original source. There is a set of invariants for each exit point, as well as a set of invariants that hold for all exit points. Look at the invariants when `top()` returns at line 79.

```
===========================================================================
StackAr.top():::EXIT79
return == this.theArray[this.topOfStack]
this.theArray == orig(this.theArray)
this.theArray[] == orig(this.theArray[])
this.topOfStack == orig(this.topOfStack)
return != null
this.topOfStack >= 0
this.theArray[this.topOfStack+1..] elements == this.theArray[-1]
===========================================================================
```

The return value is never null, and is equal to the array element at index `topOfStack`. The top of the stack is at least 0. The array, the elements of the array, and `topOfStack` are not modified by this method — this method is an "observer". The last invariant is not particularly interesting.

3.1.4 Understanding DynComp

To get a sense of how DynComp helps eliminate uninteresting output, take a look at the invariants for the exit point at line 32 of the `createItem(int)` method.

```
===========================================================================
DataStructures.StackArTester.createItem(int):::EXIT32
this.phase == 4
return.getClass() == int[].class
this.s.topOfStack < orig(i)
this.phase < orig(i)
this.phase != size(this.s.theArray[])
this.maxPhase < orig(i)
===========================================================================
```

The value of `phase` is always less than the value of `i`. While this is true for the observed executions, it is not a helpful invariant, since `phase` and `i` represent different abstract types; `i` is a number to be pushed onto the stack, while `phase` is used for program flow control. Although they are both `int`s, comparing the two is not meaningful, so this invariant, among others, is omitted from the output when Daikon is run with DynComp.

```
===========================================================================
DataStructures.StackArTester.createItem(int):::EXIT32
this.phase == 4
return.getClass() == int[].class
this.s.topOfStack < orig(i)
===========================================================================
```

3.1.5 A second Java example

A second example is located in the '`examples/java-examples/QueueAr`' subdirectory. Run this sample using the following steps:

- Compile

```
cd examples/java-examples/QueueAr
javac -g DataStructures/*.java
```

- Trace file generation and invariant detection

  ```
  java daikon.Chicory --daikon DataStructures.QueueArTester
  ```

Alternately, you can split the second command into two parts:

- Trace file generation

  ```
  java daikon.Chicory DataStructures.QueueArTester
  ```

- Invariant detection

  ```
  java daikon.Daikon QueueArTester.dtrace.gz
  ```

3.2 Detecting invariants in C/C++ programs

In order to detect invariants over C or C++ programs, you must first install a C/C++ front end (instrumenter). You can choose Kvasir (see Section 7.3 [Kvasir], page 100) or Mangel-Wurzel (see Section 7.4 [Mangel-Wurzel], page 122). We recommend the use of Kvasir, and this section gives examples using Kvasir.

To use the C/C++ front end Kvasir with your program, first make sure that your program has been compiled with DWARF-2 format debugging information, such as by giving the '`-gdwarf-2`' flag to GCC when compiling. Then, run your program as usual, but prepend `kvasir-dtrace` to the command line.

For more information about Kvasir, including more detailed documentation on its command-line options, see Section 7.3 [Kvasir], page 100.

3.2.1 C examples

The Daikon distribution comes with several example C programs in the '`examples/c-examples`' directory to enable users to become familiar with running Daikon on C programs.

To detect invariants for a program with Kvasir, you need to perform two basic tasks: run the program under Kvasir to create a data trace file (steps 1–2), and run Daikon over the data trace file to produce invariants (steps 3–4). The following instructions are for the wordplay example, which is a program for finding anagrams.

1. Change to the directory containing the program.

   ```
   cd $DAIKONDIR/examples/c-examples/wordplay
   ```

2. Compile the program with DWARF-2 debugging information enabled (and all optimizations disabled).

   ```
   gcc -gdwarf-2 wordplay.c -o wordplay
   ```

   Kvasir can also be used for programs constructed by compiling a number of '`.c`' files separately, and then linking them together; in such a program, specify '`-gdwarf-2`' when compiling each source file containing code you wish to see invariants about.

3. Run the program just as you normally would, but prepend `kvasir-dtrace` to the command line.

   ```
   kvasir-dtrace ./wordplay -f words.txt 'daikon dynamic invariant detector'
   ```

   Any options to the program can be specified as usual; here, for instance, we give commands to look for anagrams of the phrase “Daikon Dynamic Invariant Detector” using words from the file '`words.txt`'.

Executing under Kvasir, the program runs normally, but Kvasir executes additional checks and collects trace information (for this reason, the program will run more slowly than usual). Kvasir creates a directory named '`daikon-output`' under the current directory, and creates the '`wordplay.dtrace`' file, which lists both variable declarations and values.

Kvasir will also print messages if it observes your program doing something with undefined effects; these may indicate bugs in your program, or they may be spurious. (If they are bugs, they can also be tracked down by using Valgrind (`http://www.valgrind.org/`) with its regular memory checking tool; if they do not appear with that tool, they are probably spurious).

4. Run Daikon on the trace file.

```
java daikon.Daikon \
     --config_option daikon.derive.Derivation.disable_derived_variables=true \
     daikon-output/wordplay.dtrace
```

The invariants are printed to standard output, and a binary representation of the invariants is written to '`wordplay.inv.gz`'. Note that the example uses a configuration option to disable the use of derived variables; it can also run without that option, but takes significantly longer.

Daikon can analyze multiple runs (executions) of the program. You can supply Daikon with multiple trace files:

```
kvasir-dtrace --dtrace-file=daikon-output/wordplay1.dtrace ./wordplay -f words.txt
kvasir-dtrace --dtrace-file=daikon-output/wordplay2.dtrace ./wordplay -f words.txt
kvasir-dtrace --dtrace-file=daikon-output/wordplay3.dtrace ./wordplay -f words.txt
java -Xmx256m daikon.Daikon daikon-output/wordplay*.dtrace
```

or, you can append information from multiple runs in a single trace file:

```
kvasir-dtrace --dtrace-file=daikon-output/wordplay-all.dtrace ./wordplay -f words.t
kvasir-dtrace --dtrace-append --dtrace-file=daikon-output/wordplay-all.dtrace ./wor
kvasir-dtrace --dtrace-append --dtrace-file=daikon-output/wordplay-all.dtrace ./wor
java -Xmx256m daikon.Daikon daikon-output/wordplay-all.dtrace
```

5. Examine the invariants. As described in Section 3.1.1 [StackAr example], page 7, there are several ways to do this:
   - Examine the output from running Daikon.
   - Use the PrintInvariants program to display the invariants.

   For help understanding the invariants, see Section 5.4 [Interpreting output], page 27.

3.2.2 Using DynComp with C programs

Optionally, the DynComp tool can be used along with Kvasir and Daikon. DynComp uses a dynamic analysis to infer which program variables can meaningfully be used together; Daikon can then use this information to restrict the invariants it considers, potentially improving both its performance and the usefulness of its results.

DynComp is enabled as an extra mode of Kvasir; when running with DynComp enabled, Kvasir produces two output files instead of one: in addition to a '`.dtrace`' file containing a trace of a particular execution, the information about what variables and functions exist in a program, along with information grouping the variables into abstract types, is stored in a file with the extension '.decls'. Both of these files must be supplied to Daikon.

For instance, to repeat the wordplay example with DynComp, first rerun kvasir-dtrace giving it the option '--with-dyncomp':

```
kvasir-dtrace --with-dyncomp \
   ./wordplay -f words.txt 'daikon dynamic invariant detector'
```

Then, supply the '.decls' file when invoking Daikon:

```
java daikon.Daikon \
     --config_option daikon.derive.Derivation.disable_derived_variables=true \
     daikon-output/wordplay.decls daikon-output/wordplay.dtrace
```

For instance, one effect of DynComp that can be seen in the wordplay example concerns the global variables `largestlet`, `rec_anag_count`, `adjacentdups`, `specfirstword`, `maxdepthspec`, and `silent`. These variables are all 0 in the sample execution (for instance, several of them correspond to command-line options that are not enabled), so without DynComp, Daikon gives the invariants that they are all equal. However, DynComp's analysis finds that the variables are of different abstract types, so it is not meaningful to compare them. When DynComp information is provided, Daikon instead gives separate invariants about the value of each variable.

3.2.3 Dealing with large examples

Since the default memory size used by Java virtual machines varies, we suggest that Daikon be run with at least 256 megabytes of memory (and perhaps much more), specified for many JVMs by the option '-Xmx256m'. For more information about specifying the memory usage for Daikon, see Section 9.1.6 [Out of memory], page 156.

Disk usage can be reduced by specifying that the front end should compress its output dtrace files.

In some cases, the time and space requirements of the examples can be reduced by reducing the length of the program run. However, Daikon's running time depends on both the length of the test run and the size of the program data (such as its use of global variables and nested data structures). The examples also demonstrate disabling derived variables, which significantly improves Daikon's performance at the cost of producing fewer invariants. For more techniques for using Daikon with large programs and long program runs, see Section 9.2 [Large dtrace files], page 159.

3.3 Detecting invariants in Perl programs

The Daikon front end for Perl is called dfepl.

Using the Perl front end is a two-pass process: first you must run the annotated program so that the runtime system can dynamically infer the kind of data stored in each variable, and then you must re-annotate and re-run the program with the added type information. This is necessary because Perl programs do not contain type declarations.

dfepl requires version 5.8 or later of Perl.

3.3.1 Instrumenting Perl programs

Perl programs must be instrumented twice. First they must be instrumented without type information. Then, once the first instrumented version has been run to produce type information, they must be instrumented again taking the type information into account.

To instrument a stand-alone Perl program, invoke dfepl with the name of the program as an argument.

```
dfepl program.pl
```

To instrument a Perl module or a collection of modules, invoke dfepl either with the name of each module, or with the name of a directory containing the modules. To instrument all the modules in the current directory, give dfepl the argument '`.`'. For instance, if the current directory contains a module `Acme::Trampoline` in '`Acme/Trampoline.pm`' and another module `Acme::Date` in '`Acme/Date.pm`', they can be annotated by either of the following two commands:

```
dfepl Acme/Trampoline.pm Acme/Date.pm
dfepl .
```

Once type information is available, run the instrumentation command again with the '`-T`' or '`-t`' options added to use the produced type information.

For more information about dfepl, see Section 7.5 [dfepl], page 133.

3.3.2 Perl examples

The Daikon distribution includes sample Perl programs suitable for use with Daikon in the '`examples/perl-examples`' directory.

Here are step-by-step instructions for examining a simple module, '`Birthday.pm`', as used by a test script '`test-bday.pl`'.

1. Change to the directory containing the '`Birthday.pm`' module.

   ```
   cd examples/perl-examples
   ```

2. Instrument the '`Birthday.pm`' file.

   ```
   dfepl Birthday.pm
   ```

   This command creates a directory '`daikon-untyped`', and puts the instrumented version of '`Birthday.pm`' into '`daikon-untyped/Birthday.pm`'. As the directory name implies, this instrumented version doesn't contain type information.

3. Run a test suite using the instrumented '`Birthday.pm`' file.

   ```
   dtype-perl test_bday.pl 10
   ```

   The '`dtype-perl`' is a script that runs Perl with the appropriate command line options to find the modules used by the Daikon Perl runtime tracing modules, and to use the instrumented versions of modules in '`daikon-untyped`' in preference to their original ones. The number 10 is an argument to the '`test_bday.pl`' script telling it to run a relatively short test.

   This will also generate a file '`daikon-instrumented/Birthday.types`' recording the type of each variable seen during the execution of the instrumented program.

4. Re-annotate the module using the type information.

   ```
   dfepl -T Birthday.pm
   ```

This step repeats step 2, except that the '-T' flag to dfepl tells it to use the type information generated in the previous step, and to put the output in the directory 'daikon-instrumented'. dfepl also converts the type information into a file 'daikon-output/Birthday.decls' containing subroutine declarations suitable for Daikon.

5. Run the full test suite with the type-instrumented 'Birthday.pm'.

```
dtrace-perl test_bday.pl 30
```

Here we run another test suite, which happens to be the same 'test_bday.pl', but running for longer. (The example will also work with a smaller number). The script `dtrace-perl` is similar to `dtype-perl` mentioned earlier, but looks for instrumented source files in 'daikon-instrumented'.

This creates 'daikon-output/test_bday-combined.dtrace', a trace file containing the values of variables at each invocation. (The filename is formed from the name of the test program, with '-combined' appended because it contains the trace information from all the instrumented modules invoked from the program).

6. Change to the 'daikon-output' directory to analyze the output.

```
cd daikon-output
```

7. Run Daikon on the trace file

```
java daikon.Daikon Birthday.decls test_bday-combined.dtrace
```

8. Examine the invariants. They are printed to standard output, and they are also saved to file 'Birthday.inv.gz', which you can manipulate with the PrintInvariants program and other Daikon tools. For example:

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

Invariants produced from Perl programs can be examined using the same tools as other Daikon invariants.

In the example above, the script 'test_bday.pl' was not itself instrumented; it was only used to test the instrumented code. The Perl front end can also be used to instrument stand-alone Perl programs. The following sequence of commands, similar to those above, show how Daikon can be used with the stand-alone program 'standalone.pl', also in the 'examples/perl-examples' directory.

```
dfepl standalone.pl
dtype-perl daikon-untyped/standalone.pl
dfepl -T standalone.pl
dtrace-perl daikon-instrumented/standalone.pl
cd daikon-output
java daikon.Daikon -o standalone.inv standalone-main.decls \
     standalone-combined.dtrace
```

Note two differences when running a stand-alone program. First, the instrumented versions of the program, in the 'daikon-untyped' or 'daikon-instrumented' directory, are run directly. Second, the declarations file is named after the package in which the subroutines were declared, but since every stand-alone program uses the `main` package, the name of the program is prepended to the '.decls' file name to avoid collisions.

3.4 Detecting invariants in Eiffel programs

CITADEL is an Eiffel front-end to the Daikon invariant detector. You can obtain Citadel and its documentation from `http://se.inf.ethz.ch/people/polikarpova/citadel.html`.

4 Running Daikon

This section describes how to run Daikon on a data trace (‘`.dtrace`’) file, and describes Daikon’s command-line options. This section assumes you have already run a front end (e.g., an instrumenter) to produce a ‘`.dtrace`’ file (and optionally ‘`.decl`’ and ‘`.spinfo`’ files); to learn more about that process, see Chapter 3 [Example usage], page 7, and see Chapter 7 [Front ends (instrumentation)], page 88.

Run the Daikon invariant detector via the command

```
java daikon.Daikon [flags] dtrace-files... [decl-files...] [spinfo-files...]
```

- The *dtrace-files* are data trace (‘`.dtrace`’) files containing variable values from an execution of the target program.
- The *decl-files* are declaration (‘`.decl`’) files containing program point declarations. Be sure to include all declaration files that are needed for the particular data trace file; the simplest way is to include every declaration file created when instrumenting the program.

  Not all Daikon front ends produce ‘`.decl`’ files, since program point declarations may also appear in ‘`.dtrace`’ files. For instance, the Chicory front end for Java (see Section 7.1 [Chicory], page 88) produces only ‘`.dtrace`’ files. If there are no ‘`.decl`’ files, then it is not necessary to include them on the command line to Daikon.
- The *spinfo-files* are splitter info (‘`.spinfo`’) files that enable detection of conditional invariants (see Section 6.2 [Conditional invariants], page 79); these are optional and may be created automatically or by hand.

The files may appear in any order; the file type is determined by whether the file name contains ‘`.decls`’, ‘`.dtrace`’, or ‘`.spinfo`’. As a special case, a file name of ‘`-`’ means to read data trace information from standard input.

The optional flags are described in the sections that follow. For further ways to control Daikon’s behavior via configuration options, see Section 6.1 [Configuration options], page 52, and see the list of options to the front end such as Chicory (see Section 7.1.1 [Chicory options], page 88) or Kvasir (see Section 7.3.2 [Kvasir options], page 101).

4.1 Options to control Daikon output

‘`--help`’ Print usage message.

‘`-o` *`inv_file`*’
: Output serialized invariants to the specified file; they can later be postprocessed, compared, etc. Default: ‘*`basename`*`.inv.gz`’ in the current directory, where the first data trace file’s basename starts with ‘*`basename`*`.dtrace`’. Default is no serialized output, if no such data trace file was supplied. If a data trace file was supplied, there is currently no way to avoid creating a serialized invariant file.

‘`--no_text_output`’
: Don’t print invariants as text output. This option may be used in conjunction with the ‘`-o`’ option.

‘`--format` *`name`*’
: Produce output in the given format. For a list of the output formats supported by Daikon, see Section 5.1 [Invariant syntax], page 22.

'--show_progress'
'--no_show_progress'
: Prints (respectively, suppresses) timing information as major portions of Daikon are executed.

'--noversion'
: Suppress the printing of version information

'--output_num_samples'
: Output numbers of values and samples for invariants and program points; this is a debugging flag. (That is, it helps you understand why Daikon produced the output that it did.)

  The "Samples breakdown" output indicates how many samples in the '.dtrace' file had a modified value ("m"), had an unmodified value ("u"), and had a missing value ("x"). The summary uses a capital letter if the sample had any of the corresponding type of variable, and a lower-case letter if it had none. These types affect statistical tests that determine whether a particular invariant (that was true over all the test runs) is printed.

  Only variables that appear in both the pre-state and the post-state (variables that are in scope at both procedure exit and entry) are eligible to be listed as modified or unmodified. This is why the list of all variables is not the union of the modified and unmodified variables.

'--files_from *filename*'
: Read a list of '.decl', '.dtrace', or '.spinfo' files from the given text file, one filename per line, as an alternative to providing them on the command line.

'--server *dirname*'
: Server mode for Daikon in which it reads files from *dirname* as they appear (sorted lexicographically) until it finds a file ending in ".end", at which point it calculates and outputs the invariants.

'--omit_from_output [0rs]'
: Omit some potentially redundant information from the serialized output file produced with '-o'. By default, the serialized output contains all of the data structures produced by Daikon while inferring invariants. Depending on the use to which the serialized output will later be put, the file can sometimes be significantly shortened by omitting information that is no longer needed. The flag should be followed by one or more characters each representing a kind of structures the can be omitted. The following characters are recognized:

  '0 (zero)'
  : Omit information about program points that were declared, but for which no samples were found in any '.dtrace' file.

  'r'
  : Omit "reflexive" invariants, those in which a variable appears more than once. Usually, such invariants are not interesting, because their meaning is duplicated by invariants with fewer variables: for instance, `x = x - x` and `y = z + z` can be expressed as `x = 0` and `y = 2 * z` instead. However, Daikon generates and uses such invariants internally to decide what invariants to create when two previously equal variables turn out to be different.

'`s`' Omit invariants that are suppressed by other invariants. "Suppression" refers to a particular optimization in which the processing of an invariant is postponed as long as certain other invariants that logically imply it hold.

For most uses of serialized output in the current version, it is safe to use the '`0`' and '`r`' omissions, but the '`s`' omission will cause subtle output changes. In many cases, the amount of space saved is modest (typically around 10%), but the savings can be more substantial for programs with many unused program points, or program points with many variables.

4.2 Options to control invariant detection

'`--conf_limit` *val*'

Set the confidence limit for justifying invariants. If the confidence level for a given invariant is larger than the limit, then Daikon outputs the invariant. This mechanism filters out invariants that are satisfied purely by chance. This is only relevant to invariants that were true in all observed samples; Daikon never outputs invariants that were ever false.

val must be between 0 and 1; the default is .99. Larger values yield stronger filtering.

Each type of invariant has its own rules for determining confidence. See the `computeConfidence` method in the Java source code for the invariant.

For example, consider the invariant *a*<*b* whose confidence computation is `1 - 1/2^`*`numsamples`*, which indicates the likelihood that the observations of *a* and *b* did not occur by chance. If there were 3 samples, and *a*<*b* on all of them, then the confidence would be 7/8 = .875. If there were 6 samples, and *a*<*b* on only 5 on them, the confidence would be 0. If there were 9 samples, and *a*<*b* on all of them, then the confidence would be 1-1/2^9 = .998.

'`--list_type` *classname*'

Indicate that the given class implements the `java.util.List` interface. The preferred mechanism for indicating such information is the `ListImplementors` section of the '`.decls`' file. See section "V1 ListImplementors declaration" in *Daikon Developer Manual*.

'`--nohierarchy`'

Avoid connecting program points in a dataflow hierarchy. For example, Daikon normally connects the `:::ENTER` program points of class methods with the class's `:::CLASS` program point, so that any invariant that holds on the `:::CLASS` program point is considered to hold true on the `:::ENTER` program point. With no hierarchy, each program point is treated independently. This is for using Daikon on applications that do not have a concept of hierarchy. It can also be useful when you wish to process unmatched enter point samples from a trace file that is missing some exit point samples.

'`--suppress_redundant`'

Suppress display of logically redundant invariants, using the Simplify automatic theorem prover. Daikon already suppresses most logically redundant output.

For example, if “x >= 0” and “x > 0” are both true, then Daikon outputs only “x > 0”. Use of the ‘`--suppress_redundant`’ option tells Daikon to use Simplify to eliminate even more redundant output, and should be used if it is important that absolutely no redundancies appear in the output.

Simplify must be separately obtained (from `http://www.hpl.hp.com/downloads/crl/jtk/`) and installed in order to take advantage of this option. Beware that Simplify can run slowly; the amount of effort Simplify exerts for each invariant can be controlled using the ‘`daikon.simplify.Session.simplify_max_iterations`’ and ‘`daikon.simplify.Session.simplify_timeout`’ configuration options.

4.3 Processing only part of the trace file

Using ‘`--ppt-select-pattern`’ and ‘`--ppt-omit-pattern`’ can save time even if there are no samples for the excluded program points, as Daikon can skip the declarations and need not initialize data structures that would be used if samples were encountered.

‘`--ppt-select-pattern=`*`ppt_regexp`*’
: Only process program points whose names match the regular expression. The ‘`--ppt-omit-pattern`’ argument takes precedence over this argument.

‘`--ppt-omit-pattern=`*`ppt_regexp`*’
: Do not process program points whose names match the regular expression. This takes precedence over the ‘`--ppt-select-pattern`’ argument.

‘`--var-select-pattern=`*`var_regexp`*’
: Only process variables (whether in the trace file or derived) whose names match the regular expression. The ‘`--var-omit-pattern`’ argument takes precedence over this argument.

‘`--var-omit-pattern=`*`var_regexp`*’
: Ignore variables (whether in the trace file or derived) whose names match the regular expression. This takes priority over the ‘`--var-select-pattern`’ argument.

The *...-omit-pattern* arguments take precedence: if a name matches an omit pattern, it is excluded. If a name does not match an omit pattern, it is tested against the select pattern (if any). If any select patterns are specified, the name must match one of the patterns in order to be included. If no select patterns are specified, then any ppt name that does not match the omit patterns is included.

All of the regular expressions used by Daikon use Java’s regular expression syntax, which is similar to Perl’s but not exactly the same. Details are available at `http://java.sun.com/j2se/1.5.0/docs/api/java/util/regex/Pattern.html#sum`. Multiple items can be matched by using the logical or operator (`|`), for example *var1*`|`*var2*`|`*var3*.

4.4 Daikon configuration options

‘`--config` *`filename`*’
: Load the configuration settings specified in the given file. See Section 6.1 [Configuration options], page 52, for details.

'--config_option *name=value*'

Specify a single configuration setting. See Section 6.1 [Configuration options], page 52, for details.

4.5 Daikon debugging options

'--dbg *category*'

'--debug' These debugging options cause output to be written to a log file (by default, to the terminal); in other words, they enable a Logger. The '--dbg *category*' option enables debugging output for a specific part of Daikon; it may be specified multiple times, permitting find-grained control over debugging output. The '--debug' option turns on all debugging flags. (This produces a lot of output!) Most categories are class or package names in the Daikon implementation, such as daikon.split or daikon.derive.binary.SequencesJoin. Only classes that check the appropriate categories are affected by the debug flags; you can determine this by looking for a call to 'Logger.getLogger' in the specific class.

'--track *class<var1,var2,var3>@ppt*'

Turns on debugging information on the specified class, variables, and program point. In contrast to the '--dbg' option, track logging follows a particular invariant through Daikon. Multiple '--track' options can be specified. Each item (class, variables, and program point) is optional. Multiple classes can be specified separated by vertical bars (|). Matching is a simple substring (not a regular expression) comparison. Each item must match in order for a printout to occur. For more information, see section "Track logging" in *Daikon Developer Manual*.

'--disc_reason *inv_class<var1,var2,...>@ppt*'

Prints all discarded invariants of class *inv_class* at the program point specified that involve exactly the variables given, as well as a short reason and discard code explaining why they were not worthy of print. Any of the three parts of the argument may be made a wildcard by excluding it. For example, '*inv_class*' and '*<var1,var2,...>@ppt*' are valid arguments. Concrete examples are 'Implication<x,y>@foo():::EXIT', 'Implication<x,y>', and '<x,y>@foo():::EXIT'. To print all discarded invariants, use the argument 'all'.

'--mem_stat'

Prints memory usage statistics into a file named 'stat.out' in the current directory.

5 Daikon output

Daikon outputs the invariants that it discovers in textual form to your terminal. This chapter describes how to interpret those invariants — in other words, what do they mean?

Daikon also creates a '.inv' file that contains the invariants in serialized (binary) form. You can use the '.inv' file to print the invariants (see Section 8.1.1 [Printing invariants], page 142) in a variety of formats, to insert the invariants in your source code (see Section 8.1.4 [Annotate], page 145), to perform run-time checking of the invariants (see Section 8.1.6 [Runtime-check instrumenter], page 147, and Section 8.1.7 [InvariantChecker], page 149), and to do various other operations. See Chapter 8 [Tools], page 142, for descriptions of such tools.

If you wish to write your own tools for processing invariants, you have two general options. You can parse Daikon's textual output, or you can write Java code that processes the '.inv' file. The '.inv' file is simply a serialized `PptMap` (`http://groups.csail.mit.edu/pag/daikon/download/jdoc/daikon/PptMap.html`) object. In addition to reading the Javadoc, you can examine how the other tools use this data structure.

5.1 Invariant syntax

Daikon can produce output in a variety of formats. Each of the format names can be specified as an argument to the '`--format`' argument of Daikon (see Section 4.1 [Options to control Daikon output], page 17), PrintInvariants (see Section 8.1.1 [Printing invariants], page 142), and Annotate (see Section 8.1.4 [Annotate], page 145). When passed on the command line, the format names are case-insensitive: '`--format JML`' and '`--format jml`' have the same effect.

You can enhance Daikon to produce output in other formats. See section "New formatting for invariants" in *Daikon Developer Manual*

Daikon format
: Daikon's default format is a mix of Java, mathematical logic, and some additional extensions. It is intended to concisely convey meaning to programmers.

DBC format
: This format produces output in the design-by-contract (DBC) format expected by Parasoft's Jtest tool (`http://www.parasoft.com`).

ESC/Java format
ESC format
: The Extended Static Checker for Java (ESC/Java) is a programming tool for finding errors in Java programs by checking annotations that are inserted in source code; for more details, see `http://www.hpl.hp.com/downloads/crl/jtk/`. Daikon's ESC/Java format (which can also be specified as ESC format) is intended for use with the original ESC/Java tool. Use Daikon's JML format for use with the ESC/Java2 tool.

Java format
: Write output as Java expressions. This means that each invariant would be a valid Java expression, if inserted at the correct program point: right after

method entry, for method entry invariants; right before method exit, for method exit invariants;, or anywhere in the code, for object invariants.

There are two exceptions. Method exit invariants that refer to prestate, such as '`x == old(x) + 1`', are output with the tag '`\old`' surrounding the prestate expression (e.g. '`x == \old(x) + 1`'. Method exit invariants that refer to the return value of the method, such as '`return == x + y`', are output with the tag '`\result`' in place of the return value (e.g. '`\result == x + y`'). These expression are obviously not valid Java code.

JML format

Produces output in JML (Java Modeling Language, `http://www.jmlspecs.org`); for details, see the JML Manual. JML format lets you use the various JML tools on Daikon invariants, including runtime assertion checking and the ESC/Java2 tool.

Simplify format

Produces output in the format expected by the Simplify automated theorem prover; for details, see the Simplify distribution.

5.2 Program points

A program point is a specific place in the source code, such as immediately before a particular line of code. Daikon's output is organized by program points.

For example, `foo():::ENTER` is the point at the entry to procedure `foo()`; the invariants at that point are the preconditions for the `foo()` method, properties that are always true when the procedure is invoked.

Likewise, `foo():::EXIT` is the program point at the procedure exit, and invariants there are postconditions. When there are multiple exit points from a procedure (for instance, because of multiple `return` statements), the different exits are differentiated by suffixing them with their line numbers; for instance, `StackAr.top():::EXIT79`. The exit point lacking a line number (in this example, `StackAr.top():::EXIT`) collects the postconditions that are true at every numbered exit point. This is an example of a program point that represents a collection of locations in the program source rather than a single location. This concept is represented in Daikon by the dataflow hierarchy, see section "Dataflow hierarchy" in *Daikon Developer Manual*.

Two other program point tags that have special meaning to Daikon's hierarchy organization are `:::OBJECT` and `:::CLASS`. The `:::OBJECT` tag indicates object invariants (sometimes called representation invariants or class invariants) over all the instance (member) fields and static fields of the class. These properties always hold for any object of the given class, from the point of view of a client or user. These properties hold at entry to and exit from every public method of the class (except not the entry to constructors, when fields are not yet initialized).

The `:::CLASS` tag is just like `:::OBJECT`, but only for static variables, which have only one value for all objects. Static fields and instance fields are often used for different purposes. Daikon's separation of the two types of fields permits programmers to see the properties over the static fields without knowing which are the static fields and pick them out of the `:::OBJECT` program point.

(By contrast, ESC/Java and JML make class invariants hold even at the entry and exit of private methods. Their designers believe that most private methods preserve the class invariant and are called only when the class invariant holds. ESC/Java and JML require an explicit "helper" annotation to indicate a private method for which the class invariant does not hold.)

The Java instrumenter Chicory selects names for program points that include an indication of the argument and return types for each method. These signatures are presented in the JVML format: one character for each primitive type ('`B`' for byte, '`C`' for character, '`Z`' for boolean (so as not to clash with byte), etc.); '`L`*`classname`*`;`' for object types; and a '`[`' prefix for each level of array nesting.

5.3 Variable names

A front end produces a trace file that associates trace variable names with values. Trace variable names need not be exactly the same as the variables in the program. The trace may contain values that are not held in any program variables; in this case, the front end must make up a name to express that value (see below for examples).

Daikon ignores variable names when inferring invariants; it uses the names only when performing output. (Thus, the only practical restriction on trace names is that the VarInfoName `parse` method must be able to parse the name.)

By convention, trace variables are similar to program variables and field accesses. For example, `w` and `x.y.z` are legal trace variables. (So are '`a[i]`', and '`a[0].next`', but these are usually handled as derived variables instead; see below.) As in languages such as Java and C, a period character represents field access and square brackets represent selecting an element of a sequence.

In addition to variables that appear in the trace file, Daikon creates additional variables, called "derived variables", by combining trace variables. For example, for any array `a` and integer `i`, Daikon creates a derived variable `a[i]`. This is not a variable in the program (and this expression might not even appear in the source code), but it may still be useful to compute invariants over this expression. For a list of derived variables and how to control Daikon's use of them, see Section 6.1.1.4 [Options to enable/disable derived variables], page 69.

Some trace variables and derived variables may represent meaningless expressions; in such a circumstance, the value is said to be nonsensical (see section "Nonsensical values" in *Daikon Developer Manual*).

The remainder of this section describes conventions for naming expressions. Those that cannot be named by simple C/Java expressions are primarily related to arrays and sequences. (In part, these special expressions are necessary because Daikon can only handle variables of scalar (integer, floating-point, boolean, String) and array-of-scalar types. Daikon cannot handle structs, classes, or multi-dimensional arrays or structures, but such data structures can be represented as scalars and arrays by choosing variable names that indicate their relationship.)

- `a[i]` array access. `a` and `i` are themselves arbitrary variable names, of array and integral type, respectively.
- `a[-1]` from-end array access. `a[-1]` denotes the last element of array `a`; it is syntactic sugar for `a[a.length-1]`.

- `a[]` array contents. For array-valued expression `a`, all of its elements, as a sequence. Simply using the expression `a` means the identity (address or hashcode) of the array, not a list of its elements. For two arrays `a` and `b`, '`a=b`' implies '`a[]=b[]`', but '`a[]=b[]`' does not imply '`a=b`'.
- `x.y`, `x->y` field access. When field access is applied to a structure/class, it has the usual meaning of selecting one field from the structure/class.

  When field access is applied to an array, it means to map the field access across the elements of the array. For example, if `a` is an array, then `a[].foo` is the sequence consisting of the `foo` fields of each of the elements of `a`. Likewise, `a[].foo.bar` contains the `bar` fields of `a[].foo`. By contrast, `a.foo` does not make sense, because one cannot ask for the `foo` field of an address, and `a[].foo[]` would be a two-dimensional array.
- As in Java, `x.getClass()` is the runtime type of `x`, which may differ from its declared type.
- `a.length` is the length (number of elements) of array `a`; this is not necessarily the number of initialized or used elements.
- `s.toString` is the string value of String `s`, namely a sequence of characters.
- `Classname.varname` static class variable. Static variables of a class have names of the form '*`classname.varname`*'
- `orig(x)` refers to the value of variable `x` upon entry to a procedure (because the procedure body might modify the value of `x`). These variables appear only at `:::EXIT` program points. Typically, `orig()` variables do not appear in the trace, but are automatically created by Daikon when it matches up `:::ENTER`*`[nn]`* and `:::EXIT` program points. See Section 5.3.1 [orig variable example], page 26.

  This variable prints as `orig` when using Daikon output format (see Section 5.1 [Invariant syntax], page 22), but may print differently in other formats (such as `\old`).
- `post(x)` refers to the value of variable `x` upon exit from a procedure. Such a value is usually written simply `x`; the `post` prefix is needed only within an `orig` expression, when the post-state value needs to be referenced. Just as `orig` may be used only in a post-state context and specifies an expression to be evaluated in the pre-state, `post` may be may be used only in a pre-state context and specifies an expression to be evaluated in the post-state. See Section 5.3.1 [orig variable example], page 26.
- `/globalVar` C global variable. In C output, global variables with external linkage are prefixed with a slash. For instance, global `/x` is distinct from procedure parameter `/x`. (In Java programs, variables can be distinguished by prefixing them with `this.` or, for class-static variables, a class name.)
- `myfile_c/staticVar` C static variable. In C output, file-static variables have names of the form '*`filename/varname`*', where periods ('`.`') in the filename are converted into underscores ('`_`'). For example, '`Global_c/x`' is the name for a file-static variable `x` declared in the file '`Global.c`').
- `myfile_c@funcname/funcStaticVar` C function-scoped static variable. In C output, for static variables which are declared within functions, an at-sign '`@`' separates the filename and the function name and then a slash separates the function name and variable name (e.g., '`Global_c@main/funcStaticVar`' for a static variable `funcStaticVar` declared within the function `main` in the file '`Global.c`').

Daikon's current front ends do not produce output for local variables, only for variables visible from outside a procedure. (Also see the '`--std-visibility`' option to Chicory, Section 7.1.1 [Chicory options], page 88.) More generally, Daikon's front ends produce output at procedure exit and entry, not within the procedure. Thus, Daikon's output forms a specification from the view of a client of a procedure. If you wish to compute invariants over local variables, you can extend one of Daikon's front ends (or request us to do so). An alternative that permits computing invariants at arbitrary locations is to call a dummy procedure, passing all the variables of interest. The dummy procedure's pre- and post-conditions will be identical and will represent the invariants at the point of call.

The array introduction operator `[]` can made Daikon variables look slightly odd, but it is intended to assist in interpreting the variables and to provide an indication that the variable name cannot be substituted directly in a program as an expression.

Each array introduction operator `[]` increases the dimensionality of the variable, and each array indexing operation `[i]` decreases it. Since all Daikon variables are scalars or one-dimensional arrays, these operators must be matched up, or have at most one more `[]` than `[i]`. (There is one exception: according to a strict interpretation of the rules, the C/Java expression `a[i]` would turn into the Daikon variable `a[][i]`, since it does not change the dimensionality of any expression it appears in. However, that would be even more confusing, and the point is to avoid confusion, so by convention Daikon front ends use just `a[i]`, not `a[][i]`. Strictly speaking, none of the `[]` operators is necessary, since a user with a perfect knowledge of the type of each program variable and field could use that to infer the type of any Daikon expression.)

5.3.1 orig variable example

This section gives an example of use of `orig()` and `post()` variables and arrays.

Suppose you have initially that (in Java syntax)

```
int i = 0;
int[] a = new int[] { 22, 23 };
int[] b = new int[] { 46, 47 };
```

and then you run the following:

```
// pre-state values at this point
a[0] = 24;
a[1] = 25
a = b;
a[0] = 48;
a[1] = 49;
i = 1;
// post-state values at this point
```

The values of various variables are as follows:

`orig(a[i]) = 22`
: The value of `a[i]` in the pre-state: {22, 23}[0]

`orig(a[])[orig(i)] = 22`
: This is the same as orig(a[i]): {22, 23}[0].

`orig(a[])[i] = 23`
: The value of `a[]` in the pre-state (which is an array object, not a reference), indexed by the post-state value of i: {22, 23}[1]

`orig(a)[orig(i)] = 24`
: `orig(a)` is the original value of the reference `a`, not `a`'s original elements: {24, 25}[0]

`orig(a)[i] = 25`
: The original pointer value of a, indexed by the post-state value of i: {24, 25}[1]

`a[orig(i)] = 48`
: In the post-state, `a` indexed by the original value of `i`: {48, 49}[0]

`a[i] = 49`
: The value of `a[i]` in the post-state.

`b = orig(b) = some hashcode`
: The identity of the array `b` has not changed.

`b[] = [48, 49]`
`orig(b[]) = [46, 47]`
: For an array `b`, '`b=orig(b)`' does not imply '`b[]=orig(b[])`'.

`orig(a[post(i)]) = 23`
: The pre-state value of `a[1]` (because the post-state value of `i` is 1): {22, 23}[1]

5.4 Interpreting Daikon output

If nothing gets printed before the '`Exiting`' line, then Daikon found no invariants. You can get a little bit more information by using the '`--output_num_samples`' flag to Daikon (see Section 4.1 [Options to control Daikon output], page 17).

Daikon's output is predicated on the assumption that all expressions that get evaluated are sensible. For instance, if Daikon prints '`a.b == 0`', then that means that if '`a.b`' is sensible (that is, '`a`' is non-null), then its value is zero. If you would like the assumptions to be printed explicitly, then set the '`daikon.Daikon.guardNulls`' configuration option (see Section 6.1.1.6 [General configuration options], page 73).

5.4.1 Redundant

By default, Daikon does not display redundant invariants — those that are implied by other invariants in the output — because such results would merely clutter the output without adding any valuable information. For instance, if Daikon reports '`x==y`', then it never also reports '`x-1==y-1`'. You can control this behavior to some extent by disabling invariant filters; See Section 5.6 [Invariant filters], page 50. (You can also print all invariants, even redundant ones, by saving the invariants to a '`.inv`' file and then using the PrintInvariants (see Section 8.1.1 [Printing invariants], page 142) or Diff (see Section 8.1.3 [Invariant Diff], page 143) programs to print the results.)

5.4.2 Equal variables

When two or more variables are always equal, any invariant that is true over one variable is true over all of the variables. Daikon prints invariants only over one variable (the leader) from the equal set.

An equality invariant is printed for each non-leader in the equal set. For example, if the variables `a`, `b`, and `c` are all equal and `a` is chosen as the leader, the printed invariants will include any invariants over `a` and the equality invariants '`a=b`' and '`a=c`'.

You can control which variables are in an equality set; section "Variable comparability" in *Daikon Developer Manual*.

5.4.3 Has only one value variables

The output '*var* `has only one value`' in Daikon's output means that every time that variable *var* was encountered, it had the same value. Daikon ordinarily reports the actual value, as in '*var* `== 22`'. Typically this means that the variable is a hashcode or address — that is, its declared type is '`hashcode`' (see section "Variable declarations" in *Daikon Developer Manual*). For example, '*var* `== 0x38E8A`' is not very illuminating, but it is still interesting that *var* was never rebound to a different object.

Note that '*var* `has only one value`' is different from saying that *var* is unmodified.

A variable might have only one value but *not* be reported as unmodified because the variable is not a parameter to a procedure — for instance, if a routine always returns the same object, or in a class invariant. A variable can be reported as unmodified but *not* have only one value because the variable is never modified during any execution of the procedure, but has different values on different invocations of the procedure.

5.5 Invariant list

The following is a list of all of the invariants that Daikon detects. Each invariant has a configuration enable switch. By default most invariants are enabled. Any that are not enabled by default are indicated below. Some invariants also have additional configuration switches that control their behavior. These are indicated below as well. See Section 6.1.1.2 [Options to enable/disable specific invariants], page 53.

AndJoiner
: This is a special invariant used internally by Daikon to represent an antecedent invariant in an implication where that antecedent consists of two invariants anded together.

CommonFloatSequence
: Represents sequences of double values that contain a common subset. Prints as '`{e1, e2, e3, ...} subset of x[]`'.

  This invariant is not enabled by default. See the configuration option '`daikon.inv.unary.sequence.CommonFloatSequence.enabled`'.

  See also the following configuration option:

  - '`daikon.inv.unary.sequence.CommonFloatSequence.hashcode_seqs`'

CommonSequence
: Represents sequences of long values that contain a common subset. Prints as '`{e1, e2, e3, ...} subset of x[]`'.

  This invariant is not enabled by default. See the configuration option '`daikon.inv.unary.sequence.CommonSequence.enabled`'.

  See also the following configuration option:

  - '`daikon.inv.unary.sequence.CommonSequence.hashcode_seqs`'

CommonStringSequence
: Represents string sequences that contain a common subset. Prints as "{s1, s2, s3, ...} subset of x[]".

  This invariant is not enabled by default. See the configuration option '`daikon.inv.unary.stringsequence.CommonStringSequence.enabled`'.

DummyInvariant
: This is a special invariant used internally by Daikon to represent invariants whose meaning Daikon doesn't understand. The only operation that can be performed on a DummyInvariant is to print it. For instance, dummy invariants can be created to correspond to splitting conditions, when no other invariant in Daikon's grammar is equivalent to the condition.

  To use dummy invariants for splitting conditions, the configuration option '`daikon.PptTopLevel.dummy_invariant_level`' must be set, and formatting information must be supplied in the splitter info file.

EltLowerBound
: Represents the invariant that each element of a sequence of long values is greater than or equal to a constant. Prints as '`x[] elements >= c`'.

  See also the following configuration options:

  - '`daikon.inv.unary.sequence.EltLowerBound.minimal_interesting`'
  - '`daikon.inv.unary.sequence.EltLowerBound.maximal_interesting`'

EltLowerBoundFloat
: Represents the invariant that each element of a sequence of double values is greater than or equal to a constant. Prints as '`x[] elements >= c`'.

  See also the following configuration options:

  - '`daikon.inv.unary.sequence.EltLowerBoundFloat.minimal_interesting`'
  - '`daikon.inv.unary.sequence.EltLowerBoundFloat.maximal_interesting`'

EltNonZero
: Represents the invariant "x != 0" where x represents all of the elements of a sequence of long. Prints as '`x[] elements != 0`'.

EltNonZeroFloat
: Represents the invariant "x != 0" where x represents all of the elements of a sequence of double. Prints as '`x[] elements != 0`'.

EltOneOf
: Represents sequences of long values where the elements of the sequence take on only a few distinct values. Prints as either '`x[] == c`' (when there is only one value), or as '`x[] one of {c1, c2, c3}`' (when there are multiple values).

  See also the following configuration options:

  - '`daikon.inv.unary.sequence.EltOneOf.size`'
  - '`daikon.inv.unary.sequence.EltOneOf.omit_hashcode_values_Simplify`'

EltOneOfFloat
: Represents sequences of double values where the elements of the sequence take on only a few distinct values. Prints as either '`x[] == c`' (when there is only one value), or as '`x[] one of {c1, c2, c3}`' (when there are multiple values).

See also the following configuration option:

- ‘daikon.inv.unary.sequence.EltOneOfFloat.size’

EltOneOfString
: Represents sequences of String values where the elements of the sequence take on only a few distinct values. Prints as either ‘x[] == c’ (when there is only one value), or as ‘x[] one of {c1, c2, c3}’ (when there are multiple values).

  See also the following configuration option:

  - ‘daikon.inv.unary.stringsequence.EltOneOfString.size’

EltRangeFloat.EqualMinusOne
: Internal invariant representing double scalars that are equal to minus one. Used for non-instantiating suppressions. Will never print since OneOf accomplishes the same thing

EltRangeFloat.EqualOne
: Internal invariant representing double scalars that are equal to one. Used for non-instantiating suppressions. Will never print since OneOf accomplishes the same thing

EltRangeFloat.EqualZero
: Internal invariant representing double scalars that are equal to zero. Used for non-instantiating suppressions. Will never print since OneOf accomplishes the same thing.

EltRangeFloat.GreaterEqual64
: Internal invariant representing double scalars that are greater than or equal to 64. Used for non-instantiating suppressions. Will never print since Bound accomplishes the same thing

EltRangeFloat.GreaterEqualZero
: Internal invariant representing double scalars that are greater than or equal to 0. Used for non-instantiating suppressions. Will never print since Bound accomplishes the same thing

EltRangeInt.BooleanVal
: Internal invariant representing longs whose values are always 0 or 1. Used for non-instantiating suppressions. Will never print since OneOf accomplishes the same thing.

EltRangeInt.Bound0_63
: Internal invariant representing longs whose values are between 0 and 63. Used for non-instantiating suppressions. Will never print since Bound accomplishes the same thing.

EltRangeInt.EqualMinusOne
: Internal invariant representing long scalars that are equal to minus one. Used for non-instantiating suppressions. Will never print since OneOf accomplishes the same thing

EltRangeInt.EqualOne
: Internal invariant representing long scalars that are equal to one. Used for non-instantiating suppressions. Will never print since OneOf accomplishes the same thing

EltRangeInt.EqualZero
: Internal invariant representing long scalars that are equal to zero. Used for non-instantiating suppressions. Will never print since OneOf accomplishes the same thing.

EltRangeInt.Even
: Invariant representing longs whose values are always even. Used for non-instantiating suppressions. Since this is not covered by the Bound or OneOf invariants it is printed.

  This invariant is not enabled by default. See the configuration option '`daikon.inv.unary.sequence.EltRangeInt.Even.enabled`'.

EltRangeInt.GreaterEqual64
: Internal invariant representing long scalars that are greater than or equal to 64. Used for non-instantiating suppressions. Will never print since Bound accomplishes the same thing

EltRangeInt.GreaterEqualZero
: Internal invariant representing long scalars that are greater than or equal to 0. Used for non-instantiating suppressions. Will never print since Bound accomplishes the same thing

EltRangeInt.PowerOfTwo
: Invariant representing longs whose values are always a power of 2 (exactly one bit is set). Used for non-instantiating suppressions. Since this is not covered by the Bound or OneOf invariants it is printed.

EltUpperBound
: Represents the invariant that each element of a sequence of long values is less than or equal to a constant. Prints as '`x[] elements <= c`'.

  See also the following configuration options:

  - '`daikon.inv.unary.sequence.EltUpperBound.minimal_interesting`'
  - '`daikon.inv.unary.sequence.EltUpperBound.maximal_interesting`'

EltUpperBoundFloat
: Represents the invariant that each element of a sequence of double values is less than or equal to a constant. Prints as '`x[] elements <= c`'.

  See also the following configuration options:

  - '`daikon.inv.unary.sequence.EltUpperBoundFloat.minimal_interesting`'
  - '`daikon.inv.unary.sequence.EltUpperBoundFloat.maximal_interesting`'

EltwiseFloatEqual
: Represents equality between adjacent elements (x[i], x[i+1]) of a double sequence. Prints as '`x[] elements are equal`'.

EltwiseFloatGreaterEqual
: Represents the invariant ">=" between adjacent elements (x[i], x[i+1]) of a double sequence. Prints as '`x[] sorted by ">="`'.

EltwiseFloatGreaterThan
: Represents the invariant ">" between adjacent elements (x[i], x[i+1]) of a double sequence. Prints as '`x[] sorted by ">"`'.

EltwiseFloatLessEqual
: Represents the invariant "<=" between adjacent elements (x[i], x[i+1]) of a double sequence. Prints as '`x[] sorted by "<="`'.

EltwiseFloatLessThan
: Represents the invariant "<" between adjacent elements (x[i], x[i+1]) of a double sequence. Prints as '`x[] sorted by "<"`'.

EltwiseIntEqual
: Represents equality between adjacent elements (x[i], x[i+1]) of a long sequence. Prints as '`x[] elements are equal`'.

EltwiseIntGreaterEqual
: Represents the invariant ">=" between adjacent elements (x[i], x[i+1]) of a long sequence. Prints as '`x[] sorted by ">="`'.

EltwiseIntGreaterThan
: Represents the invariant ">" between adjacent elements (x[i], x[i+1]) of a long sequence. Prints as '`x[] sorted by ">"`'.

EltwiseIntLessEqual
: Represents the invariant "<=" between adjacent elements (x[i], x[i+1]) of a long sequence. Prints as '`x[] sorted by "<="`'.

EltwiseIntLessThan
: Represents the invariant "<" between adjacent elements (x[i], x[i+1]) of a long sequence. Prints as '`x[] sorted by "<"`'.

Equality
: Keeps track of sets of variables that are equal. Other invariants are instantiated for only one member of the Equality set, the leader. If variables '`x`', '`y`', and '`z`' are members of the Equality set and '`x`' is chosen as the leader, then the Equality will internally convert into binary comparison invariants that print as '`x == y`' and '`x == z`'.

FloatEqual
: Represents an invariant of "==" between two double scalars.

FloatGreaterEqual
: Represents an invariant of ">=" between two double scalars.

FloatGreaterThan
: Represents an invariant of ">" between two double scalars.

FloatLessEqual
: Represents an invariant of "<=" between two double scalars.

FloatLessThan
: Represents an invariant of `"<"` between two double scalars.

FloatNonEqual
: Represents an invariant of `"!="` between two double scalars.

FunctionBinary.BitwiseAndLong_{xyz, yxz, zxy}
: Represents the invariant '`x = BitwiseAnd (y, z)`' over three long scalars. Since the function is symmetric, only the permutations xyz, yxz, and zxy are checked.

FunctionBinary.BitwiseOrLong_{xyz, yxz, zxy}
: Represents the invariant '`x = BitwiseOr (y, z)`' over three long scalars. Since the function is symmetric, only the permutations xyz, yxz, and zxy are checked.

FunctionBinary.BitwiseXorLong_{xyz, yxz, zxy}
: Represents the invariant '`x = BitwiseXor (y, z)`' over three long scalars. Since the function is symmetric, only the permutations xyz, yxz, and zxy are checked.

FunctionBinary.DivisionLong_{xyz, xzy, yxz, yzx, zxy, zyx}
: Represents the invariant '`x = Division (y, z)`' over three long scalars. Since the function is non-symmetric, all six permutations of the variables are checked.

FunctionBinary.GcdLong_{xyz, yxz, zxy}
: Represents the invariant '`x = Gcd (y, z)`' over three long scalars. Since the function is symmetric, only the permutations xyz, yxz, and zxy are checked.

FunctionBinary.LogicalAndLong_{xyz, yxz, zxy}
: Represents the invariant '`x = LogicalAnd (y, z)`' over three long scalars. Since the function is symmetric, only the permutations xyz, yxz, and zxy are checked.

FunctionBinary.LogicalOrLong_{xyz, yxz, zxy}
: Represents the invariant '`x = LogicalOr (y, z)`' over three long scalars. Since the function is symmetric, only the permutations xyz, yxz, and zxy are checked.

FunctionBinary.LogicalXorLong_{xyz, yxz, zxy}
: Represents the invariant '`x = LogicalXor (y, z)`' over three long scalars. Since the function is symmetric, only the permutations xyz, yxz, and zxy are checked.

FunctionBinary.LshiftLong_{xyz, xzy, yxz, yzx, zxy, zyx}
: Represents the invariant '`x = Lshift (y, z)`' over three long scalars. Since the function is non-symmetric, all six permutations of the variables are checked.

FunctionBinary.MaximumLong_{xyz, yxz, zxy}
: Represents the invariant '`x = Maximum (y, z)`' over three long scalars. Since the function is symmetric, only the permutations xyz, yxz, and zxy are checked.

FunctionBinary.MinimumLong_{xyz, yxz, zxy}
: Represents the invariant '`x = Minimum (y, z)`' over three long scalars. Since the function is symmetric, only the permutations xyz, yxz, and zxy are checked.

FunctionBinary.ModLong_{xyz, xzy, yxz, yzx, zxy, zyx}
: Represents the invariant '`x = Mod (y, z)`' over three long scalars. Since the function is non-symmetric, all six permutations of the variables are checked.

FunctionBinary.MultiplyLong_{xyz, yxz, zxy}
: Represents the invariant '`x = Multiply (y, z)`' over three long scalars. Since the function is symmetric, only the permutations xyz, yxz, and zxy are checked.

FunctionBinary.PowerLong_{xyz, xzy, yxz, yzx, zxy, zyx}
: Represents the invariant '`x = Power (y, z)`' over three long scalars. Since the function is non-symmetric, all six permutations of the variables are checked.

FunctionBinary.RshiftSignedLong_{xyz, xzy, yxz, yzx, zxy, zyx}
: Represents the invariant '`x = RshiftSigned (y, z)`' over three long scalars. Since the function is non-symmetric, all six permutations of the variables are checked.

FunctionBinary.RshiftUnsignedLong_{xyz, xzy, yxz, yzx, zxy, zyx}
: Represents the invariant '`x = RshiftUnsigned (y, z)`' over three long scalars. Since the function is non-symmetric, all six permutations of the variables are checked.

FunctionBinaryFloat.DivisionDouble_{xyz, xzy, yxz, yzx, zxy, zyx}
: Represents the invariant '`x = Division (y, z)`' over three double scalars. Since the function is non-symmetric, all six permutations of the variables are checked.

FunctionBinaryFloat.MaximumDouble_{xyz, yxz, zxy}
: Represents the invariant '`x = Maximum (y, z)`' over three double scalars. Since the function is symmetric, only the permutations xyz, yxz, and zxy are checked.

FunctionBinaryFloat.MinimumDouble_{xyz, yxz, zxy}
: Represents the invariant '`x = Minimum (y, z)`' over three double scalars. Since the function is symmetric, only the permutations xyz, yxz, and zxy are checked.

FunctionBinaryFloat.MultiplyDouble_{xyz, yxz, zxy}
: Represents the invariant '`x = Multiply (y, z)`' over three double scalars. Since the function is symmetric, only the permutations xyz, yxz, and zxy are checked.

GuardingImplication
: This is a special implication invariant that guards any invariants that are over variables that are sometimes missing. For example, if the invariant '`a.x = 0`' is true, the guarded implication is '`a != null => a.x = 0`'.

Implication
: The Implication invariant class is used internally within Daikon to handle invariants that are only true when certain other conditions are also true (splitting).

IntEqual
: Represents an invariant of "==" between two long scalars.

IntGreaterEqual
: Represents an invariant of ">=" between two long scalars.

IntGreaterThan
: Represents an invariant of ">" between two long scalars.

IntLessEqual
: Represents an invariant of "<=" between two long scalars.

IntLessThan
: Represents an invariant of "<" between two long scalars.

IntNonEqual
: Represents an invariant of "!=" between two long scalars.

  See also the following configuration option:

  - '`daikon.inv.binary.twoScalar.IntNonEqual.integral_only`'

IsPointer
: IsPointer is an invariant that heuristically determines whether an integer represents a pointer (a 32-bit memory address). Since both a 32-bit integer and an address have the same representation, sometimes a a pointer can be mistaken for an integer. When this happens, several scalar invariants are computed for integer variables. Most of them would not make any sense for pointers. Determining whether a 32-bit variable is a pointer can thus spare the computation of many irrelevant invariants.

  The basic approach is to discard the invariant if any values that are not valid pointers are encountered. By default values between -100,000 and 100,00 (except 0) are considered to be invalid pointers. This approach has been experimentally confirmed on Windows x86 executables.

  This invariant is not enabled by default. See the configuration option '`daikon.inv.unary.scalar.IsPointer.enabled`'.

LinearBinary
: Represents a Linear invariant between two long scalars '`x`' and '`y`', of the form '`ax + by + c = 0`'. The constants '`a`', '`b`' and '`c`' are mutually relatively prime, and the constant '`a`' is always positive.

LinearBinaryFloat
: Represents a Linear invariant between two double scalars '`x`' and '`y`', of the form '`ax + by + c = 0`'. The constants '`a`', '`b`' and '`c`' are mutually relatively prime, and the constant '`a`' is always positive.

LinearTernary
: Represents a Linear invariant over three long scalars '`x`', '`y`', and '`z`', of the form '`ax + by + cz + d = 0`'. The constants '`a`', '`b`', '`c`', and '`d`' are mutually relatively prime, and the constant '`a`' is always positive.

LinearTernaryFloat
: Represents a Linear invariant over three double scalars '`x`', '`y`', and '`z`', of the form '`ax + by + cz + d = 0`'. The constants '`a`', '`b`', '`c`', and '`d`' are mutually relatively prime, and the constant '`a`' is always positive.

LowerBound
: Represents the invariant '`x >= c`', where '`c`' is a constant and '`x`' is a long scalar.

  See also the following configuration options:

  - '`daikon.inv.unary.scalar.LowerBound.minimal_interesting`'
  - '`daikon.inv.unary.scalar.LowerBound.maximal_interesting`'

LowerBoundFloat
: Represents the invariant '`x >= c`', where '`c`' is a constant and '`x`' is a double scalar.

See also the following configuration options:

- 'daikon.inv.unary.scalar.LowerBoundFloat.minimal_interesting'
- 'daikon.inv.unary.scalar.LowerBoundFloat.maximal_interesting'

Member
: Represents long scalars that are always members of a sequence of long values. Prints as 'x in y[]' where 'x' is a long scalar and 'y[]' is a sequence of long.

MemberFloat
: Represents double scalars that are always members of a sequence of double values. Prints as 'x in y[]' where 'x' is a double scalar and 'y[]' is a sequence of double.

MemberString
: Represents String scalars that are always members of a sequence of String values. Prints as 'x in y[]' where 'x' is a String scalar and 'y[]' is a sequence of String.

Modulus
: Represents the invariant 'x == r (mod m)' where 'x' is a long scalar variable, 'r' is the (constant) remainder, and 'm' is the (constant) modulus.

  This invariant is not enabled by default. See the configuration option 'daikon.inv.unary.scalar.Modulus.enabled'.

NoDuplicates
: Represents sequences of long that contain no duplicate elements. Prints as 'x[] contains no duplicates'.

  This invariant is not enabled by default. See the configuration option 'daikon.inv.unary.sequence.NoDuplicates.enabled'.

NoDuplicatesFloat
: Represents sequences of double that contain no duplicate elements. Prints as 'x[] contains no duplicates'.

  This invariant is not enabled by default. See the configuration option 'daikon.inv.unary.sequence.NoDuplicatesFloat.enabled'.

NonModulus
: Represents long scalars that are never equal to r (mod m) where all other numbers in the same range (i.e., all the values that x doesn't take from min(x) to max(x)) are equal to r (mod m). Prints as 'x != r (mod m)', where 'r' is the remainder and 'm' is the modulus.

  This invariant is not enabled by default. See the configuration option 'daikon.inv.unary.scalar.NonModulus.enabled'.

NonZero
: Represents long scalars that are non-zero. Prints as 'x != 0', or as 'x != null' for pointer types.

NonZeroFloat
: Represents double scalars that are non-zero. Prints as 'x != 0'.

NumericFloat.Divides
: Represents the divides without remainder invariant between two double scalars. Prints as 'x % y == 0'.

NumericFloat.Square
: Represents the square invariant between two double scalars. Prints as '`x = y**2`'.

NumericFloat.ZeroTrack
: Represents the zero tracks invariant between two double scalars; that is, when '`x`' is zero, '`y`' is also zero. Prints as '`x = 0 ==> y = 0`'.

  This invariant is not enabled by default. See the configuration option '`daikon.inv.binary.twoScalar.NumericFloat.ZeroTrack.enabled`'.

NumericInt.BitwiseAndZero
: Represents the BitwiseAnd == 0 invariant between two long scalars; that is, '`x`' and '`y`' have no bits in common. Prints as '`x & y == 0`'.

  This invariant is not enabled by default. See the configuration option '`daikon.inv.binary.twoScalar.NumericInt.BitwiseAndZero.enabled`'.

NumericInt.BitwiseComplement
: Represents the bitwise complement invariant between two long scalars. Prints as '`x = ~y`'.

  This invariant is not enabled by default. See the configuration option '`daikon.inv.binary.twoScalar.NumericInt.BitwiseComplement.enabled`'.

NumericInt.BitwiseSubset
: Represents the bitwise subset invariant between two long scalars; that is, the bits of '`y`' are a subset of the bits of '`x`'. Prints as '`x = y | x`'.

  This invariant is not enabled by default. See the configuration option '`daikon.inv.binary.twoScalar.NumericInt.BitwiseSubset.enabled`'.

NumericInt.Divides
: Represents the divides without remainder invariant between two long scalars. Prints as '`x % y == 0`'.

NumericInt.ShiftZero
: Represents the ShiftZero invariant between two long scalars; that is, '`x`' right-shifted by '`y`' is always zero. Prints as '`x >> y = 0`'.

  This invariant is not enabled by default. See the configuration option '`daikon.inv.binary.twoScalar.NumericInt.ShiftZero.enabled`'.

NumericInt.Square
: Represents the square invariant between two long scalars. Prints as '`x = y**2`'.

NumericInt.ZeroTrack
: Represents the zero tracks invariant between two long scalars; that is, when '`x`' is zero, '`y`' is also zero. Prints as '`x = 0 ==> y = 0`'.

  This invariant is not enabled by default. See the configuration option '`daikon.inv.binary.twoScalar.NumericInt.ZeroTrack.enabled`'.

OneOfFloat
: Represents double variables that take on only a few distinct values. Prints as either '`x == c`' (when there is only one value) or as '`x one of {c1, c2, c3}`' (when there are multiple values).

See also the following configuration option:

- 'daikon.inv.unary.scalar.OneOfFloat.size'

OneOfFloatSequence
: Represents double[] variables that take on only a few distinct values. Prints as either 'x == c' (when there is only one value) or as 'x one of {c1, c2, c3}' (when there are multiple values).

  See also the following configuration option:

  - 'daikon.inv.unary.sequence.OneOfFloatSequence.size'

OneOfScalar
: Represents long scalars that take on only a few distinct values. Prints as either 'x == c' (when there is only one value), 'x one of {c1, c2, c3}' (when there are multiple values), or 'x has only one value' (when 'x' is a hashcode (pointer) - this is because the numerical value of the hashcode (pointer) is uninteresting).

  See also the following configuration options:

  - 'daikon.inv.unary.scalar.OneOfScalar.size'
  - 'daikon.inv.unary.scalar.OneOfScalar.omit_hashcode_values_Simplify'

OneOfSequence
: Represents long[] variables that take on only a few distinct values. Prints as either 'x == c' (when there is only one value) or as 'x one of {c1, c2, c3}' (when there are multiple values).

  See also the following configuration options:

  - 'daikon.inv.unary.sequence.OneOfSequence.size'
  - 'daikon.inv.unary.sequence.OneOfSequence.omit_hashcode_values_Simplify'

OneOfString
: Represents String variables that take on only a few distinct values. Prints as either 'x == c' (when there is only one value) or as 'x one of {c1, c2, c3}' (when there are multiple values).

  See also the following configuration option:

  - 'daikon.inv.unary.string.OneOfString.size'

OneOfStringSequence
: Represents String[] variables that take on only a few distinct values. Prints as either 'x == c' (when there is only one value) or as 'x one of {c1, c2, c3}' (when there are multiple values).

  See also the following configuration option:

  - 'daikon.inv.unary.stringsequence.OneOfStringSequence.size'

PairwiseFloatEqual
: Represents an invariant between corresponding elements of two sequences of double values. The length of the sequences must match for the invariant to hold. A comparison is made over each '(x[i], y[i])' pair. Thus, 'x[0]' is compared to 'y[0]', 'x[1]' to 'y[1]', and so forth. Prints as 'x[] == y[]'.

PairwiseFloatGreaterEqual
: Represents an invariant between corresponding elements of two sequences of double values. The length of the sequences must match for the invariant to hold. A comparison is made over each '`(x[i], y[i])`' pair. Thus, '`x[0]`' is compared to '`y[0]`', '`x[1]`' to '`y[1]`', and so forth. Prints as '`x[] >= y[]`'.

PairwiseFloatGreaterThan
: Represents an invariant between corresponding elements of two sequences of double values. The length of the sequences must match for the invariant to hold. A comparison is made over each '`(x[i], y[i])`' pair. Thus, '`x[0]`' is compared to '`y[0]`', '`x[1]`' to '`y[1]`', and so forth. Prints as '`x[] > y[]`'.

PairwiseFloatLessEqual
: Represents an invariant between corresponding elements of two sequences of double values. The length of the sequences must match for the invariant to hold. A comparison is made over each '`(x[i], y[i])`' pair. Thus, '`x[0]`' is compared to '`y[0]`', '`x[1]`' to '`y[1]`', and so forth. Prints as '`x[] <= y[]`'.

PairwiseFloatLessThan
: Represents an invariant between corresponding elements of two sequences of double values. The length of the sequences must match for the invariant to hold. A comparison is made over each '`(x[i], y[i])`' pair. Thus, '`x[0]`' is compared to '`y[0]`', '`x[1]`' to '`y[1]`', and so forth. Prints as '`x[] < y[]`'.

PairwiseIntEqual
: Represents an invariant between corresponding elements of two sequences of long values. The length of the sequences must match for the invariant to hold. A comparison is made over each '`(x[i], y[i])`' pair. Thus, '`x[0]`' is compared to '`y[0]`', '`x[1]`' to '`y[1]`', and so forth. Prints as '`x[] == y[]`'.

PairwiseIntGreaterEqual
: Represents an invariant between corresponding elements of two sequences of long values. The length of the sequences must match for the invariant to hold. A comparison is made over each '`(x[i], y[i])`' pair. Thus, '`x[0]`' is compared to '`y[0]`', '`x[1]`' to '`y[1]`', and so forth. Prints as '`x[] >= y[]`'.

PairwiseIntGreaterThan
: Represents an invariant between corresponding elements of two sequences of long values. The length of the sequences must match for the invariant to hold. A comparison is made over each '`(x[i], y[i])`' pair. Thus, '`x[0]`' is compared to '`y[0]`', '`x[1]`' to '`y[1]`', and so forth. Prints as '`x[] > y[]`'.

PairwiseIntLessEqual
: Represents an invariant between corresponding elements of two sequences of long values. The length of the sequences must match for the invariant to hold. A comparison is made over each '`(x[i], y[i])`' pair. Thus, '`x[0]`' is compared to '`y[0]`', '`x[1]`' to '`y[1]`', and so forth. Prints as '`x[] <= y[]`'.

PairwiseIntLessThan
: Represents an invariant between corresponding elements of two sequences of long values. The length of the sequences must match for the invariant to hold.

A comparison is made over each '(x[i], y[i])' pair. Thus, 'x[0]' is compared to 'y[0]', 'x[1]' to 'y[1]', and so forth. Prints as 'x[] < y[]'.

PairwiseLinearBinary
: Represents a linear invariant (i.e., 'y = ax + b') between the corresponding elements of two sequences of long values. Each '(x[i], y[i])' pair is examined. Thus, 'x[0]' is compared to 'y[0]', 'x[1]' to 'y[1]' and so forth. Prints as 'y[] = a * x[] + b'.

PairwiseLinearBinaryFloat
: Represents a linear invariant (i.e., 'y = ax + b') between the corresponding elements of two sequences of double values. Each '(x[i], y[i])' pair is examined. Thus, 'x[0]' is compared to 'y[0]', 'x[1]' to 'y[1]' and so forth. Prints as 'y[] = a * x[] + b'.

PairwiseNumericFloat.Divides
: Represents the divides without remainder invariant between corresponding elements of two sequences of double. Prints as 'x[] % y[] == 0'.

PairwiseNumericFloat.Square
: Represents the square invariant between corresponding elements of two sequences of double. Prints as 'x[] = y[]**2'.

PairwiseNumericFloat.ZeroTrack
: Represents the zero tracks invariant between corresponding elements of two sequences of double; that is, when 'x[]' is zero, 'y[]' is also zero. Prints as 'x[] = 0 ==> y[] = 0'.

  This invariant is not enabled by default. See the configuration option 'daikon.inv.binary.twoSequence.PairwiseNumericFloat.ZeroTrack.enabled'.

PairwiseNumericInt.BitwiseAndZero
: Represents the BitwiseAnd == 0 invariant between corresponding elements of two sequences of long; that is, 'x[]' and 'y[]' have no bits in common. Prints as 'x[] & y[] == 0'.

  This invariant is not enabled by default. See the configuration option 'daikon.inv.binary.twoSequence.PairwiseNumericInt.BitwiseAndZero.enabled'.

PairwiseNumericInt.BitwiseComplement
: Represents the bitwise complement invariant between corresponding elements of two sequences of long. Prints as 'x[] = ~y[]'.

  This invariant is not enabled by default. See the configuration option 'daikon.inv.binary.twoSequence.PairwiseNumericInt.BitwiseComplement.enabled'.

PairwiseNumericInt.BitwiseSubset
: Represents the bitwise subset invariant between corresponding elements of two sequences of long; that is, the bits of 'y[]' are a subset of the bits of 'x[]'. Prints as 'x[] = y[] | x[]'.

  This invariant is not enabled by default. See the configuration option 'daikon.inv.binary.twoSequence.PairwiseNumericInt.BitwiseSubset.enabled'.

PairwiseNumericInt.Divides
: Represents the divides without remainder invariant between corresponding elements of two sequences of long. Prints as '`x[] % y[] == 0`'.

PairwiseNumericInt.ShiftZero
: Represents the ShiftZero invariant between corresponding elements of two sequences of long; that is, '`x[]`' right-shifted by '`y[]`' is always zero. Prints as '`x[] >> y[] = 0`'.

  This invariant is not enabled by default. See the configuration option '`daikon.inv.binary.twoSequence.PairwiseNumericInt.ShiftZero.enabled`'.

PairwiseNumericInt.Square
: Represents the square invariant between corresponding elements of two sequences of long. Prints as '`x[] = y[]**2`'.

PairwiseNumericInt.ZeroTrack
: Represents the zero tracks invariant between corresponding elements of two sequences of long; that is, when '`x[]`' is zero, '`y[]`' is also zero. Prints as '`x[] = 0 ==> y[] = 0`'.

  This invariant is not enabled by default. See the configuration option '`daikon.inv.binary.twoSequence.PairwiseNumericInt.ZeroTrack.enabled`'.

PairwiseString.SubString
: Represents the substring invariant between corresponding elements of two sequences of String. Prints as '`x[] is a substring of y[]`'.

  This invariant is not enabled by default. See the configuration option '`daikon.inv.binary.twoSequence.PairwiseString.SubString.enabled`'.

PairwiseStringEqual
: Represents an invariant between corresponding elements of two sequences of String values. The length of the sequences must match for the invariant to hold. A comparison is made over each '`(x[i], y[i])`' pair. Thus, '`x[0]`' is compared to '`y[0]`', '`x[1]`' to '`y[1]`', and so forth. Prints as '`x[] == y[]`'.

PairwiseStringGreaterEqual
: Represents an invariant between corresponding elements of two sequences of String values. The length of the sequences must match for the invariant to hold. A comparison is made over each '`(x[i], y[i])`' pair. Thus, '`x[0]`' is compared to '`y[0]`', '`x[1]`' to '`y[1]`', and so forth. Prints as '`x[] >= y[]`'.

PairwiseStringGreaterThan
: Represents an invariant between corresponding elements of two sequences of String values. The length of the sequences must match for the invariant to hold. A comparison is made over each '`(x[i], y[i])`' pair. Thus, '`x[0]`' is compared to '`y[0]`', '`x[1]`' to '`y[1]`', and so forth. Prints as '`x[] > y[]`'.

PairwiseStringLessEqual
: Represents an invariant between corresponding elements of two sequences of String values. The length of the sequences must match for the invariant to hold. A comparison is made over each '`(x[i], y[i])`' pair. Thus, '`x[0]`' is compared to '`y[0]`', '`x[1]`' to '`y[1]`', and so forth. Prints as '`x[] <= y[]`'.

PairwiseStringLessThan
: Represents an invariant between corresponding elements of two sequences of String values. The length of the sequences must match for the invariant to hold. A comparison is made over each '(x[i], y[i])' pair. Thus, 'x[0]' is compared to 'y[0]', 'x[1]' to 'y[1]', and so forth. Prints as 'x[] < y[]'.

Positive
: Represents the invariant 'x > 0' where 'x' is a long scalar. This exists only as an example for the purposes of the manual. It isn't actually used (it is replaced by the more general invariant LowerBound).

PrintableString
: Represents a string that contains only printable ascii characters (values 32 through 126 plus 9 (tab)

  This invariant is not enabled by default. See the configuration option 'daikon.inv.unary.string.PrintableString.enabled'.

RangeFloat.EqualMinusOne
: Internal invariant representing double scalars that are equal to minus one. Used for non-instantiating suppressions. Will never print since OneOf accomplishes the same thing

RangeFloat.EqualOne
: Internal invariant representing double scalars that are equal to one. Used for non-instantiating suppressions. Will never print since OneOf accomplishes the same thing

RangeFloat.EqualZero
: Internal invariant representing double scalars that are equal to zero. Used for non-instantiating suppressions. Will never print since OneOf accomplishes the same thing.

RangeFloat.GreaterEqual64
: Internal invariant representing double scalars that are greater than or equal to 64. Used for non-instantiating suppressions. Will never print since Bound accomplishes the same thing

RangeFloat.GreaterEqualZero
: Internal invariant representing double scalars that are greater than or equal to 0. Used for non-instantiating suppressions. Will never print since Bound accomplishes the same thing

RangeInt.BooleanVal
: Internal invariant representing longs whose values are always 0 or 1. Used for non-instantiating suppressions. Will never print since OneOf accomplishes the same thing.

RangeInt.Bound0_63
: Internal invariant representing longs whose values are between 0 and 63. Used for non-instantiating suppressions. Will never print since Bound accomplishes the same thing.

RangeInt.EqualMinusOne
: Internal invariant representing long scalars that are equal to minus one. Used for non-instantiating suppressions. Will never print since OneOf accomplishes the same thing

RangeInt.EqualOne
: Internal invariant representing long scalars that are equal to one. Used for non-instantiating suppressions. Will never print since OneOf accomplishes the same thing

RangeInt.EqualZero
: Internal invariant representing long scalars that are equal to zero. Used for non-instantiating suppressions. Will never print since OneOf accomplishes the same thing.

RangeInt.Even
: Invariant representing longs whose values are always even. Used for non-instantiating suppressions. Since this is not covered by the Bound or OneOf invariants it is printed.

  This invariant is not enabled by default. See the configuration option '`daikon.inv.unary.scalar.RangeInt.Even.enabled`'.

RangeInt.GreaterEqual64
: Internal invariant representing long scalars that are greater than or equal to 64. Used for non-instantiating suppressions. Will never print since Bound accomplishes the same thing

RangeInt.GreaterEqualZero
: Internal invariant representing long scalars that are greater than or equal to 0. Used for non-instantiating suppressions. Will never print since Bound accomplishes the same thing

RangeInt.PowerOfTwo
: Invariant representing longs whose values are always a power of 2 (exactly one bit is set). Used for non-instantiating suppressions. Since this is not covered by the Bound or OneOf invariants it is printed.

Reverse
: Represents two sequences of long where one is in the reverse order of the other. Prints as '`x[] is the reverse of y[]`'.

ReverseFloat
: Represents two sequences of double where one is in the reverse order of the other. Prints as '`x[] is the reverse of y[]`'.

SeqFloatEqual
: Represents an invariant between a double scalar and a a sequence of double values. Prints as '`x[] elements == y`' where '`x`' is a double sequence and '`y`' is a double scalar.

SeqFloatGreaterEqual
: Represents an invariant between a double scalar and a a sequence of double values. Prints as '`x[] elements >= y`' where '`x`' is a double sequence and '`y`' is a double scalar.

SeqFloatGreaterThan

Represents an invariant between a double scalar and a a sequence of double values. Prints as ‘`x[] elements > y`’ where ‘`x`’ is a double sequence and ‘`y`’ is a double scalar.

SeqFloatLessEqual

Represents an invariant between a double scalar and a a sequence of double values. Prints as ‘`x[] elements <= y`’ where ‘`x`’ is a double sequence and ‘`y`’ is a double scalar.

SeqFloatLessThan

Represents an invariant between a double scalar and a a sequence of double values. Prints as ‘`x[] elements < y`’ where ‘`x`’ is a double sequence and ‘`y`’ is a double scalar.

SeqIndexFloatEqual

Represents an invariant over sequences of double values between the index of an element of the sequence and the element itself. Prints as ‘`x[i] == i`’.

This invariant is not enabled by default. See the configuration option ‘`daikon.inv.unary.sequence.SeqIndexFloatEqual.enabled`’.

SeqIndexFloatGreaterEqual

Represents an invariant over sequences of double values between the index of an element of the sequence and the element itself. Prints as ‘`x[i] >= i`’.

This invariant is not enabled by default. See the configuration option ‘`daikon.inv.unary.sequence.SeqIndexFloatGreaterEqual.enabled`’.

SeqIndexFloatGreaterThan

Represents an invariant over sequences of double values between the index of an element of the sequence and the element itself. Prints as ‘`x[i] > i`’.

This invariant is not enabled by default. See the configuration option ‘`daikon.inv.unary.sequence.SeqIndexFloatGreaterThan.enabled`’.

SeqIndexFloatLessEqual

Represents an invariant over sequences of double values between the index of an element of the sequence and the element itself. Prints as ‘`x[i] <= i`’.

This invariant is not enabled by default. See the configuration option ‘`daikon.inv.unary.sequence.SeqIndexFloatLessEqual.enabled`’.

SeqIndexFloatLessThan

Represents an invariant over sequences of double values between the index of an element of the sequence and the element itself. Prints as ‘`x[i] < i`’.

This invariant is not enabled by default. See the configuration option ‘`daikon.inv.unary.sequence.SeqIndexFloatLessThan.enabled`’.

SeqIndexFloatNonEqual

Represents an invariant over sequences of double values between the index of an element of the sequence and the element itself. Prints as ‘`x[i] != i`’.

This invariant is not enabled by default. See the configuration option ‘`daikon.inv.unary.sequence.SeqIndexFloatNonEqual.enabled`’.

SeqIndexIntEqual
: Represents an invariant over sequences of long values between the index of an element of the sequence and the element itself. Prints as '`x[i] == i`'.

  This invariant is not enabled by default. See the configuration option '`daikon.inv.unary.sequence.SeqIndexIntEqual.enabled`'.

SeqIndexIntGreaterEqual
: Represents an invariant over sequences of long values between the index of an element of the sequence and the element itself. Prints as '`x[i] >= i`'.

  This invariant is not enabled by default. See the configuration option '`daikon.inv.unary.sequence.SeqIndexIntGreaterEqual.enabled`'.

SeqIndexIntGreaterThan
: Represents an invariant over sequences of long values between the index of an element of the sequence and the element itself. Prints as '`x[i] > i`'.

  This invariant is not enabled by default. See the configuration option '`daikon.inv.unary.sequence.SeqIndexIntGreaterThan.enabled`'.

SeqIndexIntLessEqual
: Represents an invariant over sequences of long values between the index of an element of the sequence and the element itself. Prints as '`x[i] <= i`'.

  This invariant is not enabled by default. See the configuration option '`daikon.inv.unary.sequence.SeqIndexIntLessEqual.enabled`'.

SeqIndexIntLessThan
: Represents an invariant over sequences of long values between the index of an element of the sequence and the element itself. Prints as '`x[i] < i`'.

  This invariant is not enabled by default. See the configuration option '`daikon.inv.unary.sequence.SeqIndexIntLessThan.enabled`'.

SeqIndexIntNonEqual
: Represents an invariant over sequences of long values between the index of an element of the sequence and the element itself. Prints as '`x[i] != i`'.

  This invariant is not enabled by default. See the configuration option '`daikon.inv.unary.sequence.SeqIndexIntNonEqual.enabled`'.

SeqIntEqual
: Represents an invariant between a long scalar and a a sequence of long values. Prints as '`x[] elements == y`' where '`x`' is a long sequence and '`y`' is a long scalar.

SeqIntGreaterEqual
: Represents an invariant between a long scalar and a a sequence of long values. Prints as '`x[] elements >= y`' where '`x`' is a long sequence and '`y`' is a long scalar.

SeqIntGreaterThan
: Represents an invariant between a long scalar and a a sequence of long values. Prints as '`x[] elements > y`' where '`x`' is a long sequence and '`y`' is a long scalar.

SeqIntLessEqual
: Represents an invariant between a long scalar and a a sequence of long values. Prints as '`x[] elements <= y`' where '`x`' is a long sequence and '`y`' is a long scalar.

SeqIntLessThan
: Represents an invariant between a long scalar and a a sequence of long values. Prints as '`x[] elements < y`' where '`x`' is a long sequence and '`y`' is a long scalar.

SeqSeqFloatEqual
: Represents invariants between two sequences of double values. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as '`x[] == y[] lexically`'.

  If order doesn't matter for each variable, then the sequences are compared to see if they are set equivalent. Prints as '`x[] == y[]`'.

  If the auxiliary information (e.g., order matters) doesn't match between two variables, then this invariant cannot apply to those variables.

SeqSeqFloatGreaterEqual
: Represents invariants between two sequences of double values. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as '`x[] >= y[] lexically`'.

  If the auxiliary information (e.g., order matters) doesn't match between two variables, then this invariant cannot apply to those variables.

SeqSeqFloatGreaterThan
: Represents invariants between two sequences of double values. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as '`x[] > y[] lexically`'.

  If the auxiliary information (e.g., order matters) doesn't match between two variables, then this invariant cannot apply to those variables.

SeqSeqFloatLessEqual
: Represents invariants between two sequences of double values. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as '`x[] <= y[] lexically`'.

  If the auxiliary information (e.g., order matters) doesn't match between two variables, then this invariant cannot apply to those variables.

SeqSeqFloatLessThan
: Represents invariants between two sequences of double values. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as '`x[] < y[] lexically`'.

  If the auxiliary information (e.g., order matters) doesn't match between two variables, then this invariant cannot apply to those variables.

SeqSeqIntEqual
: Represents invariants between two sequences of long values. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as '`x[] == y[] lexically`'.

If order doesn't matter for each variable, then the sequences are compared to see if they are set equivalent. Prints as '`x[] == y[]`'.

If the auxiliary information (e.g., order matters) doesn't match between two variables, then this invariant cannot apply to those variables.

SeqSeqIntGreaterEqual

Represents invariants between two sequences of long values. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as '`x[] >= y[] lexically`'.

If the auxiliary information (e.g., order matters) doesn't match between two variables, then this invariant cannot apply to those variables.

SeqSeqIntGreaterThan

Represents invariants between two sequences of long values. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as '`x[] > y[] lexically`'.

If the auxiliary information (e.g., order matters) doesn't match between two variables, then this invariant cannot apply to those variables.

SeqSeqIntLessEqual

Represents invariants between two sequences of long values. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as '`x[] <= y[] lexically`'.

If the auxiliary information (e.g., order matters) doesn't match between two variables, then this invariant cannot apply to those variables.

SeqSeqIntLessThan

Represents invariants between two sequences of long values. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as '`x[] < y[] lexically`'.

If the auxiliary information (e.g., order matters) doesn't match between two variables, then this invariant cannot apply to those variables.

SeqSeqStringEqual

Represents invariants between two sequences of String values. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as '`x[] == y[] lexically`'.

If order doesn't matter for each variable, then the sequences are compared to see if they are set equivalent. Prints as '`x[] == y[]`'.

If the auxiliary information (e.g., order matters) doesn't match between two variables, then this invariant cannot apply to those variables.

SeqSeqStringGreaterEqual

Represents invariants between two sequences of String values. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as '`x[] >= y[] lexically`'.

If the auxiliary information (e.g., order matters) doesn't match between two variables, then this invariant cannot apply to those variables.

SeqSeqStringGreaterThan

Represents invariants between two sequences of String values. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as '`x[] > y[] lexically`'.

If the auxiliary information (e.g., order matters) doesn't match between two variables, then this invariant cannot apply to those variables.

SeqSeqStringLessEqual

Represents invariants between two sequences of String values. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as '`x[] <= y[] lexically`'.

If the auxiliary information (e.g., order matters) doesn't match between two variables, then this invariant cannot apply to those variables.

SeqSeqStringLessThan

Represents invariants between two sequences of String values. If order matters for each variable (which it does by default), then the sequences are compared lexically. Prints as '`x[] < y[] lexically`'.

If the auxiliary information (e.g., order matters) doesn't match between two variables, then this invariant cannot apply to those variables.

StdString.SubString

Represents the substring invariant between two String scalars. Prints as '`x is a substring of y`'.

This invariant is not enabled by default. See the configuration option '`daikon.inv.binary.twoString.StdString.SubString.enabled`'.

StringEqual

Represents an invariant of `"=="` between two String scalars.

StringGreaterEqual

Represents an invariant of `">="` between two String scalars.

StringGreaterThan

Represents an invariant of `">"` between two String scalars.

StringLessEqual

Represents an invariant of `"<="` between two String scalars.

StringLessThan

Represents an invariant of `"<"` between two String scalars.

StringNonEqual

Represents an invariant of `"!="` between two String scalars.

SubSequence

Represents two sequences of long values where one sequence is a subsequence of the other. Prints as '`x[] is a subsequence of y[]`'.

This invariant is not enabled by default. See the configuration option '`daikon.inv.binary.twoSequence.SubSequence.enabled`'.

SubSequenceFloat
: Represents two sequences of double values where one sequence is a subsequence of the other. Prints as '`x[] is a subsequence of y[]`'.

  This invariant is not enabled by default. See the configuration option '`daikon.inv.binary.twoSequence.SubSequenceFloat.enabled`'.

SubSet
: Represents two sequences of long values where one of the sequences is a subset of the other; that is each element of one sequence appears in the other. Prints as either '`x[] is a subset of y[]`' or as '`x[] is a superset of y[]`'.

  This invariant is not enabled by default. See the configuration option '`daikon.inv.binary.twoSequence.SubSet.enabled`'.

SubSetFloat
: Represents two sequences of double values where one of the sequences is a subset of the other; that is each element of one sequence appears in the other. Prints as either '`x[] is a subset of y[]`' or as '`x[] is a superset of y[]`'.

  This invariant is not enabled by default. See the configuration option '`daikon.inv.binary.twoSequence.SubSetFloat.enabled`'.

SuperSequence
: Represents two sequences of long values where one sequence is a subsequence of the other. Prints as '`x[] is a subsequence of y[]`'.

  This invariant is not enabled by default. See the configuration option '`daikon.inv.binary.twoSequence.SuperSequence.enabled`'.

SuperSequenceFloat
: Represents two sequences of double values where one sequence is a subsequence of the other. Prints as '`x[] is a subsequence of y[]`'.

  This invariant is not enabled by default. See the configuration option '`daikon.inv.binary.twoSequence.SuperSequenceFloat.enabled`'.

SuperSet
: Represents two sequences of long values where one of the sequences is a subset of the other; that is each element of one sequence appears in the other. Prints as either '`x[] is a subset of y[]`' or as '`x[] is a superset of y[]`'.

  This invariant is not enabled by default. See the configuration option '`daikon.inv.binary.twoSequence.SuperSet.enabled`'.

SuperSetFloat
: Represents two sequences of double values where one of the sequences is a subset of the other; that is each element of one sequence appears in the other. Prints as either '`x[] is a subset of y[]`' or as '`x[] is a superset of y[]`'.

  This invariant is not enabled by default. See the configuration option '`daikon.inv.binary.twoSequence.SuperSetFloat.enabled`'.

UpperBound
: Represents the invariant '`x <= c`', where '`c`' is a constant and '`x`' is a long scalar.

  See also the following configuration options:

  - '`daikon.inv.unary.scalar.UpperBound.minimal_interesting`'
  - '`daikon.inv.unary.scalar.UpperBound.maximal_interesting`'

UpperBoundFloat

Represents the invariant '`x <= c`', where '`c`' is a constant and '`x`' is a double scalar.

See also the following configuration options:

- '`daikon.inv.unary.scalar.UpperBoundFloat.minimal_interesting`'
- '`daikon.inv.unary.scalar.UpperBoundFloat.maximal_interesting`'

5.6 Invariant filters

Invariant filters are used to suppress the printing of invariants that are true, but not considered "interesting" — usually because the invariants are considered obvious or redundant in a given context.

The following is a list of the invariant filters that Daikon supports. Each of these filters has a corresponding configuration enable switch; by default, all filters are enabled. See Section 6.1.1.1 [Options to enable/disable filters], page 52, for details.

- DerivedParameterFilter: suppress parameter-derived postcondition invariants

  This filter suppresses invariants at procedure exit points that are uninteresting because they refer to prestate variables derived from pass-by-value parameters. For example, suppose that `param` is a parameter to a Java method. If `param` itself is modified, that change won't be visible to a caller, so it's uninteresting to print. If `param` points to an object, and that object is changed, that is visible, but only if `param` hasn't changed; otherwise, the invariant would report a change in some object other than the one that was passed in.

- ObviousFilter: suppress obvious invariants

  This filter suppresses invariants because they are obvious from looking at other invariants. Some examples are:

  - If '`size(args[])==0`' is shown, then '`size(args[])-1==-1`' is obvious and will not be displayed by default.
  - If '`this.topOfStack < size(this.theArray[])-1`' is shown, then '`this.topOfStack < size(this.theArray[])`' is obvious and will not be displayed by default.

- OnlyConstantVariablesFilter: suppress invariants containing only constants

  This filter suppresses comparison invariants in which all of the variables being compared were observed to be constant. In the current version of Daikon, most such invariants are not even created in the first place, because constants are detected on an early pass over the data. However, Daikon will note that all of the invariants that had any particular constant value were also equal to each other: such invariants will be suppressed by this filter.

- ParentFilter: filter invariants that match a parent program point invariant

  A controlled invariant is an invariant that is "controlled" — or implied — by a parent program point in the dataflow hierarchy. For example, for Java instrumented code each class is associated with an object program point, which contain invariants that are found at the entry and exit of all public methods. So in addition to the usual program points such as `StackAr.StackAr(int):::ENTER` and `StackAr.isEmpty():::EXIT48`, daikon outputs invariants for the artificial program point `StackAr:::OBJECT`. The invariants for `StackAr:::OBJECT` control the invariants for `StackAr.StackAr(int):::ENTER` and `StackAr.isEmpty():::EXIT48`, because the former imply the latter. Because of this redundancy, controlled invariants are not displayed by default. Note that if for some reason, the controlling invariant is not displayed (for example, because it's unjustified), then the controlled invariant *will* be displayed.

- SimplifyFilter: eliminate redundant invariants using Simplify

  Daikon contains built-in test that remove most redundant (logically implied) invariants from its output; see

  Daikon can use the Simplify theorem-prover to eliminate even more implied invariants than Daikon's built-in tests are able to eliminate. Simplify must be separately obtained (from `http://www.hpl.hp.com/downloads/crl/jtk/`) and installed in order to take advantage of this filter.

  If you don't also specify the '`--suppress_redundant`' command-line option (see Section 4.2 [Options to control invariant detection], page 19) to enable Simplify processing, this filter doesn't do anything.

- UnjustifiedFilter: suppress unjustified invariants

  For every invariant, Daikon estimates the probability of that invariant happening by chance. If that probability is less than the limit, then the invariant is deemed to be an actual invariant, not just a chance occurrence. Currently the limit is .01%. So by default, only invariants with probabilities of less than .01% are shown. See the '`--conf_limit`' option (see Section 4.2 [Options to control invariant detection], page 19).

- UnmodifiedVariableEqualityFilter: suppress invariants that merely indicate that a variable was unmodified

  This filter is only useful for ESC output.

6 Enhancing Daikon output

6.1 Configuration options

Many aspects of Daikon's behavior can be controlled by setting various configuration parameters. These configuration parameters control which invariants are checked and reported, the statistical tests for invariants, which derived variables are created, and more.

The configuration options are set by creating a configuration file and supplying it to Daikon on the command line using the '`--config` *filename*' option. Daikon reads all supplied configuration files in order, overriding the defaults. You may wish to use the supplied example configuration file '`daikon/java/daikon/config/example-settings.txt`' as an example when creating your own configuration files. (If you did not download Daikon's sources, you must extract the example from '`daikon.jar`' to read it.)

You can also control Daikon's output via its command-line options (see Chapter 4 [Running Daikon], page 17) and via the command-line options to its front ends such as Chicory (see Section 7.1.1 [Chicory options], page 88) or Kvasir (see Section 7.3.2 [Kvasir options], page 101).

You may also specify a configuration setting directly on the command line, using the '`--config_option` *name=value*' option.

The configuration options are different from the debugging flags '`--debug`' and '`--dbg` *category*' (see Section 4.5 [Daikon debugging options], page 21). The debugging flags permit Daikon to produce debugging output, but they do not affect the invariants that Daikon computes.

6.1.1 List of configuration options

This is a list of all Daikon configuration options. The configuration option name contains the Java class in which it is defined. (In the Daikon source code, the configuration value is stored in a variable whose name contains a `dkconfig_` prefix, but that should be irrelevant to users.) To learn more about a specific invariant or derived variable than appears in this manual, see its source code.

6.1.1.1 Options to enable/disable filters

These configuration options enable or disable filters that suppress printing of certain invariants. Invariants are filtered if they are found to be true but are considered uninteresting or redundant. See Section 5.6 [Invariant filters], page 50, for more information.

'`daikon.inv.filter.DerivedParameterFilter.enabled`'
: Boolean. If true, DerivedParameterFilter is initially turned on. The default value is 'true'.

'`daikon.inv.filter.ObviousFilter.enabled`'
: Boolean. If true, ObviousFilter is initially turned on. The default value is 'true'.

'`daikon.inv.filter.OnlyConstantVariablesFilter.enabled`'
: Boolean. If true, OnlyConstantVariablesFilter is initially turned on. The default value is 'true'.

'`daikon.inv.filter.ParentFilter.enabled`'
: Boolean. If true, ParentFilter is initially turned on. The default value is 'true'.

'`daikon.inv.filter.SimplifyFilter.enabled`'
: Boolean. If true, SimplifyFilter is initially turned on. The default value is 'true'.

'`daikon.inv.filter.UnjustifiedFilter.enabled`'
: Boolean. If true, UnjustifiedFilter is initially turned on. The default value is 'true'.

'`daikon.inv.filter.UnmodifiedVariableEqualityFilter.enabled`'
: Boolean. If true, UnmodifiedVariableEqualityFilter is initially turned on. The default value is 'true'.

6.1.1.2 Options to enable/disable specific invariants

These options control whether Daikon looks for specific kinds of invariants. See Section 5.5 [Invariant list], page 28, for more information about the corresponding invariants.

'`daikon.inv.binary.sequenceScalar.Member.enabled`'
: Boolean. True iff Member invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.sequenceScalar.MemberFloat.enabled`'
: Boolean. True iff Member invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.sequenceScalar.SeqFloatEqual.enabled`'
: Boolean. True iff SeqFloatEqual invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.sequenceScalar.SeqFloatGreaterEqual.enabled`'
: Boolean. True iff SeqFloatGreaterEqual invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.sequenceScalar.SeqFloatGreaterThan.enabled`'
: Boolean. True iff SeqFloatGreaterThan invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.sequenceScalar.SeqFloatLessEqual.enabled`'
: Boolean. True iff SeqFloatLessEqual invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.sequenceScalar.SeqFloatLessThan.enabled`'
: Boolean. True iff SeqFloatLessThan invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.sequenceScalar.SeqIntEqual.enabled`'
: Boolean. True iff SeqIntEqual invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.sequenceScalar.SeqIntGreaterEqual.enabled`'
: Boolean. True iff SeqIntGreaterEqual invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.sequenceScalar.SeqIntGreaterThan.enabled`'
Boolean. True iff SeqIntGreaterThan invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.sequenceScalar.SeqIntLessEqual.enabled`'
Boolean. True iff SeqIntLessEqual invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.sequenceScalar.SeqIntLessThan.enabled`'
Boolean. True iff SeqIntLessThan invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.sequenceString.MemberString.enabled`'
Boolean. True iff Member invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoScalar.FloatEqual.enabled`'
Boolean. True iff FloatEqual invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoScalar.FloatGreaterEqual.enabled`'
Boolean. True iff FloatGreaterEqual invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoScalar.FloatGreaterThan.enabled`'
Boolean. True iff FloatGreaterThan invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoScalar.FloatLessEqual.enabled`'
Boolean. True iff FloatLessEqual invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoScalar.FloatLessThan.enabled`'
Boolean. True iff FloatLessThan invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoScalar.FloatNonEqual.enabled`'
Boolean. True iff FloatNonEqual invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoScalar.IntEqual.enabled`'
Boolean. True iff IntEqual invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoScalar.IntGreaterEqual.enabled`'
Boolean. True iff IntGreaterEqual invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoScalar.IntGreaterThan.enabled`'
Boolean. True iff IntGreaterThan invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoScalar.IntLessEqual.enabled`'
Boolean. True iff IntLessEqual invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoScalar.IntLessThan.enabled`'
: Boolean. True iff IntLessThan invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoScalar.IntNonEqual.enabled`'
: Boolean. True iff IntNonEqual invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoScalar.LinearBinary.enabled`'
: Boolean. True iff LinearBinary invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoScalar.LinearBinaryFloat.enabled`'
: Boolean. True iff LinearBinary invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoScalar.NumericFloat.Divides.enabled`'
: Boolean. True iff divides invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoScalar.NumericFloat.Square.enabled`'
: Boolean. True iff square invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoScalar.NumericFloat.ZeroTrack.enabled`'
: Boolean. True iff zero-track invariants should be considered. The default value is 'false'.

'`daikon.inv.binary.twoScalar.NumericInt.BitwiseAndZero.enabled`'
: Boolean. True iff BitwiseAndZero invariants should be considered. The default value is 'false'.

'`daikon.inv.binary.twoScalar.NumericInt.BitwiseComplement.enabled`'
: Boolean. True iff bitwise complement invariants should be considered. The default value is 'false'.

'`daikon.inv.binary.twoScalar.NumericInt.BitwiseSubset.enabled`'
: Boolean. True iff bitwise subset invariants should be considered. The default value is 'false'.

'`daikon.inv.binary.twoScalar.NumericInt.Divides.enabled`'
: Boolean. True iff divides invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoScalar.NumericInt.ShiftZero.enabled`'
: Boolean. True iff ShiftZero invariants should be considered. The default value is 'false'.

'`daikon.inv.binary.twoScalar.NumericInt.Square.enabled`'
: Boolean. True iff square invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoScalar.NumericInt.ZeroTrack.enabled`'
: Boolean. True iff zero-track invariants should be considered. The default value is 'false'.

'`daikon.inv.binary.twoSequence.PairwiseFloatEqual.enabled`'
Boolean. True iff PairwiseIntComparison invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoSequence.PairwiseFloatGreaterEqual.enabled`'
Boolean. True iff PairwiseIntComparison invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoSequence.PairwiseFloatGreaterThan.enabled`'
Boolean. True iff PairwiseIntComparison invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoSequence.PairwiseFloatLessEqual.enabled`'
Boolean. True iff PairwiseIntComparison invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoSequence.PairwiseFloatLessThan.enabled`'
Boolean. True iff PairwiseIntComparison invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoSequence.PairwiseIntEqual.enabled`'
Boolean. True iff PairwiseIntComparison invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoSequence.PairwiseIntGreaterEqual.enabled`'
Boolean. True iff PairwiseIntComparison invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoSequence.PairwiseIntGreaterThan.enabled`'
Boolean. True iff PairwiseIntComparison invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoSequence.PairwiseIntLessEqual.enabled`'
Boolean. True iff PairwiseIntComparison invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoSequence.PairwiseIntLessThan.enabled`'
Boolean. True iff PairwiseIntComparison invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoSequence.PairwiseLinearBinary.enabled`'
Boolean. True iff PairwiseLinearBinary invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoSequence.PairwiseLinearBinaryFloat.enabled`'
Boolean. True iff PairwiseLinearBinary invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoSequence.PairwiseNumericFloat.Divides.enabled`'
Boolean. True iff divides invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoSequence.PairwiseNumericFloat.Square.enabled`'
Boolean. True iff square invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoSequence.PairwiseNumericFloat.ZeroTrack.enabled`'
: Boolean. True iff zero-track invariants should be considered. The default value is 'false'.

'`daikon.inv.binary.twoSequence.PairwiseNumericInt.BitwiseAndZero.enabled`'
: Boolean. True iff BitwiseAndZero invariants should be considered. The default value is 'false'.

'`daikon.inv.binary.twoSequence.PairwiseNumericInt.BitwiseComplement.enabled`'
: Boolean. True iff bitwise complement invariants should be considered. The default value is 'false'.

'`daikon.inv.binary.twoSequence.PairwiseNumericInt.BitwiseSubset.enabled`'
: Boolean. True iff bitwise subset invariants should be considered. The default value is 'false'.

'`daikon.inv.binary.twoSequence.PairwiseNumericInt.Divides.enabled`'
: Boolean. True iff divides invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoSequence.PairwiseNumericInt.ShiftZero.enabled`'
: Boolean. True iff ShiftZero invariants should be considered. The default value is 'false'.

'`daikon.inv.binary.twoSequence.PairwiseNumericInt.Square.enabled`'
: Boolean. True iff square invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoSequence.PairwiseNumericInt.ZeroTrack.enabled`'
: Boolean. True iff zero-track invariants should be considered. The default value is 'false'.

'`daikon.inv.binary.twoSequence.PairwiseString.SubString.enabled`'
: Boolean. True iff SubString invariants should be considered. The default value is 'false'.

'`daikon.inv.binary.twoSequence.PairwiseStringEqual.enabled`'
: Boolean. True iff PairwiseIntComparison invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoSequence.PairwiseStringGreaterEqual.enabled`'
: Boolean. True iff PairwiseIntComparison invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoSequence.PairwiseStringGreaterThan.enabled`'
: Boolean. True iff PairwiseIntComparison invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoSequence.PairwiseStringLessEqual.enabled`'
: Boolean. True iff PairwiseIntComparison invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoSequence.PairwiseStringLessThan.enabled`'
: Boolean. True iff PairwiseIntComparison invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoSequence.Reverse.enabled`'
: Boolean. True iff Reverse invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoSequence.ReverseFloat.enabled`'
: Boolean. True iff Reverse invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoSequence.SeqSeqFloatEqual.enabled`'
: Boolean. True iff SeqSeqFloatEqual invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoSequence.SeqSeqFloatGreaterEqual.enabled`'
: Boolean. True iff SeqSeqFloatGreaterEqual invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoSequence.SeqSeqFloatGreaterThan.enabled`'
: Boolean. True iff SeqSeqFloatGreaterThan invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoSequence.SeqSeqFloatLessEqual.enabled`'
: Boolean. True iff SeqSeqFloatLessEqual invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoSequence.SeqSeqFloatLessThan.enabled`'
: Boolean. True iff SeqSeqFloatLessThan invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoSequence.SeqSeqIntEqual.enabled`'
: Boolean. True iff SeqSeqIntEqual invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoSequence.SeqSeqIntGreaterEqual.enabled`'
: Boolean. True iff SeqSeqIntGreaterEqual invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoSequence.SeqSeqIntGreaterThan.enabled`'
: Boolean. True iff SeqSeqIntGreaterThan invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoSequence.SeqSeqIntLessEqual.enabled`'
: Boolean. True iff SeqSeqIntLessEqual invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoSequence.SeqSeqIntLessThan.enabled`'
: Boolean. True iff SeqSeqIntLessThan invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoSequence.SeqSeqStringEqual.enabled`'
: Boolean. True iff SeqSeqStringEqual invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoSequence.SeqSeqStringGreaterEqual.enabled`'
: Boolean. True iff SeqSeqStringGreaterEqual invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoSequence.SeqSeqStringGreaterThan.enabled`'
: Boolean. True iff SeqSeqStringGreaterThan invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoSequence.SeqSeqStringLessEqual.enabled`'
: Boolean. True iff SeqSeqStringLessEqual invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoSequence.SeqSeqStringLessThan.enabled`'
: Boolean. True iff SeqSeqStringLessThan invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoSequence.SubSequence.enabled`'
: Boolean. True iff SubSequence invariants should be considered. The default value is 'false'.

'`daikon.inv.binary.twoSequence.SubSequenceFloat.enabled`'
: Boolean. True iff SubSequence invariants should be considered. The default value is 'false'.

'`daikon.inv.binary.twoSequence.SubSet.enabled`'
: Boolean. True iff SubSet invariants should be considered. The default value is 'false'.

'`daikon.inv.binary.twoSequence.SubSetFloat.enabled`'
: Boolean. True iff SubSet invariants should be considered. The default value is 'false'.

'`daikon.inv.binary.twoSequence.SuperSequence.enabled`'
: Boolean. True iff SubSequence invariants should be considered. The default value is 'false'.

'`daikon.inv.binary.twoSequence.SuperSequenceFloat.enabled`'
: Boolean. True iff SubSequence invariants should be considered. The default value is 'false'.

'`daikon.inv.binary.twoSequence.SuperSet.enabled`'
: Boolean. True iff SubSet invariants should be considered. The default value is 'false'.

'`daikon.inv.binary.twoSequence.SuperSetFloat.enabled`'
: Boolean. True iff SubSet invariants should be considered. The default value is 'false'.

'`daikon.inv.binary.twoString.StdString.SubString.enabled`'
: Boolean. True iff SubString invariants should be considered. The default value is 'false'.

'`daikon.inv.binary.twoString.StringEqual.enabled`'
: Boolean. True iff StringEqual invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoString.StringGreaterEqual.enabled`'
: Boolean. True iff StringGreaterEqual invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoString.StringGreaterThan.enabled`'
: Boolean. True iff StringGreaterThan invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoString.StringLessEqual.enabled`'
: Boolean. True iff StringLessEqual invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoString.StringLessThan.enabled`'
: Boolean. True iff StringLessThan invariants should be considered. The default value is 'true'.

'`daikon.inv.binary.twoString.StringNonEqual.enabled`'
: Boolean. True iff StringNonEqual invariants should be considered. The default value is 'true'.

'`daikon.inv.ternary.threeScalar.FunctionBinary.enabled`'
: Boolean. True if FunctionBinary invariants should be considered. The default value is 'false'.

'`daikon.inv.ternary.threeScalar.FunctionBinaryFloat.enabled`'
: Boolean. True if FunctionBinaryFloat invariants should be considered. The default value is 'false'.

'`daikon.inv.ternary.threeScalar.LinearTernary.enabled`'
: Boolean. True iff LinearTernary invariants should be considered. The default value is 'true'.

'`daikon.inv.ternary.threeScalar.LinearTernaryFloat.enabled`'
: Boolean. True iff LinearTernary invariants should be considered. The default value is 'true'.

'`daikon.inv.unary.scalar.IsPointer.enabled`'
: Boolean. True iff IsPointer invariants should be considered. The default value is 'false'.

'`daikon.inv.unary.scalar.LowerBound.enabled`'
: Boolean. True iff LowerBound invariants should be considered. The default value is 'true'.

'`daikon.inv.unary.scalar.LowerBoundFloat.enabled`'
: Boolean. True iff LowerBoundFloat invariants should be considered. The default value is 'true'.

'`daikon.inv.unary.scalar.Modulus.enabled`'
: Boolean. True iff Modulus invariants should be considered. The default value is 'false'.

'`daikon.inv.unary.scalar.NonModulus.enabled`'
: Boolean. True iff NonModulus invariants should be considered. The default value is 'false'.

'`daikon.inv.unary.scalar.NonZero.enabled`'
: Boolean. True iff NonZero invariants should be considered. The default value is 'true'.

'`daikon.inv.unary.scalar.NonZeroFloat.enabled`'
: Boolean. True iff NonZeroFloat invariants should be considered. The default value is 'true'.

'`daikon.inv.unary.scalar.OneOfFloat.enabled`'
: Boolean. True iff OneOf invariants should be considered. The default value is 'true'.

'`daikon.inv.unary.scalar.OneOfScalar.enabled`'
: Boolean. True iff OneOf invariants should be considered. The default value is 'true'.

'`daikon.inv.unary.scalar.Positive.enabled`'
: Boolean. True iff Positive invariants should be considered. The default value is 'true'.

'`daikon.inv.unary.scalar.RangeInt.Even.enabled`'
: Boolean. True if Even invariants should be considered. The default value is 'false'.

'`daikon.inv.unary.scalar.RangeInt.PowerOfTwo.enabled`'
: Boolean. True if PowerOfTwo invariants should be considered. The default value is 'true'.

'`daikon.inv.unary.scalar.UpperBound.enabled`'
: Boolean. True iff UpperBound invariants should be considered. The default value is 'true'.

'`daikon.inv.unary.scalar.UpperBoundFloat.enabled`'
: Boolean. True iff UpperBoundFloat invariants should be considered. The default value is 'true'.

'`daikon.inv.unary.sequence.CommonFloatSequence.enabled`'
: Boolean. True iff CommonSequence invariants should be considered. The default value is 'false'.

'`daikon.inv.unary.sequence.CommonSequence.enabled`'
: Boolean. True iff CommonSequence invariants should be considered. The default value is 'false'.

'`daikon.inv.unary.sequence.EltLowerBound.enabled`'
: Boolean. True iff EltLowerBound invariants should be considered. The default value is 'true'.

'`daikon.inv.unary.sequence.EltLowerBoundFloat.enabled`'
: Boolean. True iff EltLowerBoundFloat invariants should be considered. The default value is 'true'.

'`daikon.inv.unary.sequence.EltNonZero.enabled`'
: Boolean. True iff EltNonZero invariants should be considered. The default value is 'true'.

'`daikon.inv.unary.sequence.EltNonZeroFloat.enabled`'
: Boolean. True iff EltNonZero invariants should be considered. The default value is 'true'.

'`daikon.inv.unary.sequence.EltOneOf.enabled`'
: Boolean. True iff OneOf invariants should be considered. The default value is 'true'.

'`daikon.inv.unary.sequence.EltOneOfFloat.enabled`'
: Boolean. True iff OneOf invariants should be considered. The default value is 'true'.

'`daikon.inv.unary.sequence.EltRangeInt.Even.enabled`'
: Boolean. True if Even invariants should be considered. The default value is 'false'.

'`daikon.inv.unary.sequence.EltRangeInt.PowerOfTwo.enabled`'
: Boolean. True if PowerOfTwo invariants should be considered. The default value is 'true'.

'`daikon.inv.unary.sequence.EltUpperBound.enabled`'
: Boolean. True iff EltUpperBound invariants should be considered. The default value is 'true'.

'`daikon.inv.unary.sequence.EltUpperBoundFloat.enabled`'
: Boolean. True iff EltUpperBoundFloat invariants should be considered. The default value is 'true'.

'`daikon.inv.unary.sequence.EltwiseFloatEqual.enabled`'
: Boolean. True iff EltwiseIntComparison invariants should be considered. The default value is 'true'.

'`daikon.inv.unary.sequence.EltwiseFloatGreaterEqual.enabled`'
: Boolean. True iff EltwiseIntComparison invariants should be considered. The default value is 'true'.

'`daikon.inv.unary.sequence.EltwiseFloatGreaterThan.enabled`'
: Boolean. True iff EltwiseIntComparison invariants should be considered. The default value is 'true'.

'`daikon.inv.unary.sequence.EltwiseFloatLessEqual.enabled`'
: Boolean. True iff EltwiseIntComparison invariants should be considered. The default value is 'true'.

'`daikon.inv.unary.sequence.EltwiseFloatLessThan.enabled`'
: Boolean. True iff EltwiseIntComparison invariants should be considered. The default value is 'true'.

'`daikon.inv.unary.sequence.EltwiseIntEqual.enabled`'
: Boolean. True iff EltwiseIntComparison invariants should be considered. The default value is 'true'.

'`daikon.inv.unary.sequence.EltwiseIntGreaterEqual.enabled`'
: Boolean. True iff EltwiseIntComparison invariants should be considered. The default value is 'true'.

'`daikon.inv.unary.sequence.EltwiseIntGreaterThan.enabled`'
: Boolean. True iff EltwiseIntComparison invariants should be considered. The default value is 'true'.

'`daikon.inv.unary.sequence.EltwiseIntLessEqual.enabled`'
: Boolean. True iff EltwiseIntComparison invariants should be considered. The default value is 'true'.

'`daikon.inv.unary.sequence.EltwiseIntLessThan.enabled`'
: Boolean. True iff EltwiseIntComparison invariants should be considered. The default value is 'true'.

'`daikon.inv.unary.sequence.NoDuplicates.enabled`'
: Boolean. True iff NoDuplicates invariants should be considered. The default value is 'false'.

'`daikon.inv.unary.sequence.NoDuplicatesFloat.enabled`'
: Boolean. True iff NoDuplicates invariants should be considered. The default value is 'false'.

'`daikon.inv.unary.sequence.OneOfFloatSequence.enabled`'
: Boolean. True iff OneOf invariants should be considered. The default value is 'true'.

'`daikon.inv.unary.sequence.OneOfSequence.enabled`'
: Boolean. True iff OneOf invariants should be considered. The default value is 'true'.

'`daikon.inv.unary.sequence.SeqIndexFloatEqual.enabled`'
: Boolean. True iff SeqIndexFloatEqual invariants should be considered. The default value is 'false'.

'`daikon.inv.unary.sequence.SeqIndexFloatGreaterEqual.enabled`'
: Boolean. True iff SeqIndexFloatGreaterEqual invariants should be considered. The default value is 'false'.

'`daikon.inv.unary.sequence.SeqIndexFloatGreaterThan.enabled`'
: Boolean. True iff SeqIndexFloatGreaterThan invariants should be considered. The default value is 'false'.

'`daikon.inv.unary.sequence.SeqIndexFloatLessEqual.enabled`'
: Boolean. True iff SeqIndexFloatLessEqual invariants should be considered. The default value is 'false'.

'`daikon.inv.unary.sequence.SeqIndexFloatLessThan.enabled`'
: Boolean. True iff SeqIndexFloatLessThan invariants should be considered. The default value is 'false'.

'`daikon.inv.unary.sequence.SeqIndexFloatNonEqual.enabled`'
: Boolean. True iff SeqIndexFloatNonEqual invariants should be considered. The default value is 'false'.

'`daikon.inv.unary.sequence.SeqIndexIntEqual.enabled`'
: Boolean. True iff SeqIndexIntEqual invariants should be considered. The default value is 'false'.

'`daikon.inv.unary.sequence.SeqIndexIntGreaterEqual.enabled`'
: Boolean. True iff SeqIndexIntGreaterEqual invariants should be considered. The default value is 'false'.

'daikon.inv.unary.sequence.SeqIndexIntGreaterThan.enabled'
: Boolean. True iff SeqIndexIntGreaterThan invariants should be considered. The default value is 'false'.

'daikon.inv.unary.sequence.SeqIndexIntLessEqual.enabled'
: Boolean. True iff SeqIndexIntLessEqual invariants should be considered. The default value is 'false'.

'daikon.inv.unary.sequence.SeqIndexIntLessThan.enabled'
: Boolean. True iff SeqIndexIntLessThan invariants should be considered. The default value is 'false'.

'daikon.inv.unary.sequence.SeqIndexIntNonEqual.enabled'
: Boolean. True iff SeqIndexIntNonEqual invariants should be considered. The default value is 'false'.

'daikon.inv.unary.string.OneOfString.enabled'
: Boolean. True iff OneOf invariants should be considered. The default value is 'true'.

'daikon.inv.unary.string.PrintableString.enabled'
: Boolean. True iff PrintableString invariants should be considered. The default value is 'false'.

'daikon.inv.unary.stringsequence.CommonStringSequence.enabled'
: Boolean. True iff CommonStringSequence invariants should be considered. The default value is 'false'.

'daikon.inv.unary.stringsequence.EltOneOfString.enabled'
: Boolean. True iff OneOf invariants should be considered. The default value is 'true'.

'daikon.inv.unary.stringsequence.OneOfStringSequence.enabled'
: Boolean. True iff OneOf invariants should be considered. The default value is 'true'.

6.1.1.3 Other invariant configuration parameters

The configuration options listed in this section parameterize the behavior of certain invariants. See Section 5.5 [Invariant list], page 28, for more information about the invariants.

'daikon.inv.Invariant.confidence_limit'
: Floating-point number between 0 and 1. Invariants are displayed only if the confidence that the invariant did not occur by chance is greater than this. (May also be set via '--conf_limit' switch to Daikon; refer to manual.) The default value is '0.99'.

'daikon.inv.Invariant.fuzzy_ratio'
: Floating-point number between 0 and 0.1, representing the maximum relative difference between two floats for fuzzy comparisons. Larger values will result in floats that are relatively farther apart being treated as equal. A value of 0 essentially disables fuzzy comparisons. Specifically, if `abs (1 - f1/f2)` is less than or equal to this value, then the two doubles (`f1` and `f2`) will be treated as equal by Daikon. The default value is '1.0E-4'.

'`daikon.inv.Invariant.simplify_define_predicates`'
: A boolean value. If true, Daikon's Simplify output (printed when the '`--format simplify`' flag is enabled, and used internally by '`--suppress_redundant`') will include new predicates representing some complex relationships in invariants, such as lexical ordering among sequences. If false, some complex relationships will appear in the output as complex quantified formulas, while others will not appear at all. When enabled, Simplify may be able to make more inferences, allowing '`--suppress_redundant`' to suppress more redundant invariants, but Simplify may also run more slowly. The default value is 'false'.

'`daikon.inv.binary.twoScalar.IntNonEqual.integral_only`'
: Boolean. True iff IntNonEqual invariants should be considered. The default value is 'true'.

'`daikon.inv.filter.DerivedVariableFilter.class_re`'
: Regular expression to match against the class name of derived variables. Invariants that contain derived variables that match will be filtered out. If null, nothing will be filtered out. The default value is 'null'.

'`daikon.inv.unary.scalar.LowerBound.maximal_interesting`'
: Long integer. Together with the corresponding `minimal_interesting` parameter, specifies the range of the computed constant that is "interesting" — the range that should be reported. For instance, setting `minimal_interesting` to -1 and `maximal_interesting` to 2 would only permit output of LowerBound invariants whose cutoff was one of (-1,0,1,2). The default value is '2'.

'`daikon.inv.unary.scalar.LowerBound.minimal_interesting`'
: Long integer. Together with the corresponding `maximal_interesting` parameter, specifies the range of the computed constant that is "interesting" — the range that should be reported. For instance, setting `minimal_interesting` to -1 and `maximal_interesting` to 2 would only permit output of LowerBound invariants whose cutoff was one of (-1,0,1,2). The default value is '-1'.

'`daikon.inv.unary.scalar.LowerBoundFloat.maximal_interesting`'
: Long integer. Together with the corresponding `minimal_interesting` parameter, specifies the range of the computed constant that is "interesting" — the range that should be reported. For instance, setting `minimal_interesting` to -1 and `maximal_interesting` to 2 would only permit output of LowerBoundFloat invariants whose cutoff was one of (-1,0,1,2). The default value is '2'.

'`daikon.inv.unary.scalar.LowerBoundFloat.minimal_interesting`'
: Long integer. Together with the corresponding `maximal_interesting` parameter, specifies the range of the computed constant that is "interesting" — the range that should be reported. For instance, setting `minimal_interesting` to -1 and `maximal_interesting` to 2 would only permit output of LowerBoundFloat invariants whose cutoff was one of (-1,0,1,2). The default value is '-1'.

'`daikon.inv.unary.scalar.OneOfFloat.size`'
: Positive integer. Specifies the maximum set size for this type of invariant (x is one of `size` items). The default value is '3'.

'`daikon.inv.unary.scalar.OneOfScalar.omit_hashcode_values_Simplify`'
: Boolean. If true, invariants describing hashcode-typed variables as having any particular value will have an artificial value substituted for the exact hashhode values. The artificial values will stay the same from run to run even if the actual hashcode values change (as long as the OneOf invariants remain the same). If false, hashcodes will be formatted as the application of a hashcode uninterpreted function to an integer representing the bit pattern of the hashcode. One might wish to omit the exact values of the hashcodes because they are usually uninteresting; this is the same reason they print in the native Daikon format, for instance, as '`var has only one value`' rather than '`var == 150924732`'. The default value is 'false'.

'`daikon.inv.unary.scalar.OneOfScalar.size`'
: Positive integer. Specifies the maximum set size for this type of invariant (x is one of `size` items). The default value is '3'.

'`daikon.inv.unary.scalar.UpperBound.maximal_interesting`'
: Long integer. Together with the corresponding `minimal_interesting` parameter, specifies the range of the computed constant that is "interesting" — the range that should be reported. For instance, setting `minimal_interesting` to -1 and `maximal_interesting` to 2 would only permit output of UpperBound invariants whose cutoff was one of (-1,0,1,2). The default value is '2'.

'`daikon.inv.unary.scalar.UpperBound.minimal_interesting`'
: Long integer. Together with the corresponding `maximal_interesting` parameter, specifies the range of the computed constant that is "interesting" — the range that should be reported. For instance, setting `minimal_interesting` to -1 and `maximal_interesting` to 2 would only permit output of UpperBound invariants whose cutoff was one of (-1,0,1,2). The default value is '-1'.

'`daikon.inv.unary.scalar.UpperBoundFloat.maximal_interesting`'
: Long integer. Together with the corresponding `minimal_interesting` parameter, specifies the range of the computed constant that is "interesting" — the range that should be reported. For instance, setting `minimal_interesting` to -1 and `maximal_interesting` to 2 would only permit output of UpperBoundFloat invariants whose cutoff was one of (-1,0,1,2). The default value is '2'.

'`daikon.inv.unary.scalar.UpperBoundFloat.minimal_interesting`'
: Long integer. Together with the corresponding `maximal_interesting` parameter, specifies the range of the computed constant that is "interesting" — the range that should be reported. For instance, setting `minimal_interesting` to -1 and `maximal_interesting` to 2 would only permit output of UpperBoundFloat invariants whose cutoff was one of (-1,0,1,2). The default value is '-1'.

'`daikon.inv.unary.sequence.CommonFloatSequence.hashcode_seqs`'
: Boolean. Set to true to consider common sequences over hashcodes (pointers). The default value is 'false'.

'`daikon.inv.unary.sequence.CommonSequence.hashcode_seqs`'
: Boolean. Set to true to consider common sequences over hashcodes (pointers). The default value is 'false'.

'daikon.inv.unary.sequence.EltLowerBound.maximal_interesting'

Long integer. Together with the corresponding minimal_interesting parameter, specifies the range of the computed constant that is "interesting" — the range that should be reported. For instance, setting minimal_interesting to -1 and maximal_interesting to 2 would only permit output of EltLowerBound invariants whose cutoff was one of (-1,0,1,2). The default value is '2'.

'daikon.inv.unary.sequence.EltLowerBound.minimal_interesting'

Long integer. Together with the corresponding maximal_interesting parameter, specifies the range of the computed constant that is "interesting" — the range that should be reported. For instance, setting minimal_interesting to -1 and maximal_interesting to 2 would only permit output of EltLowerBound invariants whose cutoff was one of (-1,0,1,2). The default value is '-1'.

'daikon.inv.unary.sequence.EltLowerBoundFloat.maximal_interesting'

Long integer. Together with the corresponding minimal_interesting parameter, specifies the range of the computed constant that is "interesting" — the range that should be reported. For instance, setting minimal_interesting to -1 and maximal_interesting to 2 would only permit output of EltLowerBoundFloat invariants whose cutoff was one of (-1,0,1,2). The default value is '2'.

'daikon.inv.unary.sequence.EltLowerBoundFloat.minimal_interesting'

Long integer. Together with the corresponding maximal_interesting parameter, specifies the range of the computed constant that is "interesting" — the range that should be reported. For instance, setting minimal_interesting to -1 and maximal_interesting to 2 would only permit output of EltLowerBoundFloat invariants whose cutoff was one of (-1,0,1,2). The default value is '-1'.

'daikon.inv.unary.sequence.EltOneOf.omit_hashcode_values_Simplify'

Boolean. If true, invariants describing hashcode-typed variables as having any particular value will have an artificial value substituted for the exact hashhode values. The artificial values will stay the same from run to run even if the actual hashcode values change (as long as the OneOf invariants remain the same). If false, hashcodes will be formatted as the application of a hashcode uninterpreted function to an integer representing the bit pattern of the hashcode. One might wish to omit the exact values of the hashcodes because they are usually uninteresting; this is the same reason they print in the native Daikon format, for instance, as 'var has only one value' rather than 'var == 150924732'. The default value is 'false'.

'daikon.inv.unary.sequence.EltOneOf.size'

Positive integer. Specifies the maximum set size for this type of invariant (x is one of size items). The default value is '3'.

'daikon.inv.unary.sequence.EltOneOfFloat.size'

Positive integer. Specifies the maximum set size for this type of invariant (x is one of size items). The default value is '3'.

'`daikon.inv.unary.sequence.EltUpperBound.maximal_interesting`'
: Long integer. Together with the corresponding `minimal_interesting` parameter, specifies the range of the computed constant that is "interesting" — the range that should be reported. For instance, setting `minimal_interesting` to -1 and `maximal_interesting` to 2 would only permit output of EltUpperBound invariants whose cutoff was one of (-1,0,1,2). The default value is '2'.

'`daikon.inv.unary.sequence.EltUpperBound.minimal_interesting`'
: Long integer. Together with the corresponding `maximal_interesting` parameter, specifies the range of the computed constant that is "interesting" — the range that should be reported. For instance, setting `minimal_interesting` to -1 and `maximal_interesting` to 2 would only permit output of EltUpperBound invariants whose cutoff was one of (-1,0,1,2). The default value is '-1'.

'`daikon.inv.unary.sequence.EltUpperBoundFloat.maximal_interesting`'
: Long integer. Together with the corresponding `minimal_interesting` parameter, specifies the range of the computed constant that is "interesting" — the range that should be reported. For instance, setting `minimal_interesting` to -1 and `maximal_interesting` to 2 would only permit output of EltUpperBoundFloat invariants whose cutoff was one of (-1,0,1,2). The default value is '2'.

'`daikon.inv.unary.sequence.EltUpperBoundFloat.minimal_interesting`'
: Long integer. Together with the corresponding `maximal_interesting` parameter, specifies the range of the computed constant that is "interesting" — the range that should be reported. For instance, setting `minimal_interesting` to -1 and `maximal_interesting` to 2 would only permit output of EltUpperBoundFloat invariants whose cutoff was one of (-1,0,1,2). The default value is '-1'.

'`daikon.inv.unary.sequence.OneOfFloatSequence.size`'
: Positive integer. Specifies the maximum set size for this type of invariant (x is one of `size` items). The default value is '3'.

'`daikon.inv.unary.sequence.OneOfSequence.omit_hashcode_values_Simplify`'
: Boolean. If true, invariants describing hashcode-typed variables as having any particular value will have an artificial value substituted for the exact hashhode values. The artificial values will stay the same from run to run even if the actual hashcode values change (as long as the OneOf invariants remain the same). If false, hashcodes will be formatted as the application of a hashcode uninterpreted function to an integer representing the bit pattern of the hashcode. One might wish to omit the exact values of the hashcodes because they are usually uninteresting; this is the same reason they print in the native Daikon format, for instance, as '`var has only one value`' rather than '`var == 150924732`'. The default value is 'false'.

'`daikon.inv.unary.sequence.OneOfSequence.size`'
: Positive integer. Specifies the maximum set size for this type of invariant (x is one of `size` items). The default value is '3'.

'`daikon.inv.unary.sequence.SingleSequence.SeqIndexDisableAll`'
: Boolean. Set to true to disable all SeqIndex invariants (SeqIndexIntEqual, SeqIndexFloatLessThan, etc). This overrides the settings of the individual SeqIndex enable configuration options. To disable only some options, the options must be disabled individually. The default value is 'false'.

'`daikon.inv.unary.string.OneOfString.size`'
: Positive integer. Specifies the maximum set size for this type of invariant (x is one of `size` items). The default value is '3'.

'`daikon.inv.unary.stringsequence.EltOneOfString.size`'
: Positive integer. Specifies the maximum set size for this type of invariant (x is one of `size` items). The default value is '3'.

'`daikon.inv.unary.stringsequence.OneOfStringSequence.size`'
: Positive integer. Specifies the maximum set size for this type of invariant (x is one of `size` items). The default value is '2'.

6.1.1.4 Options to enable/disable derived variables

These options control whether Daikon looks for invariants involving certain forms of derived variables. Also see Section 5.3 [Variable names], page 24.

'`daikon.derive.Derivation.disable_derived_variables`'
: Boolean. If true, Daikon will not create any derived variables. Derived variables, which are combinations of variables that appeared in the program, like `array[index]` if `array` and `index` appeared, can increase the number of properties Daikon finds, especially over sequences. However, derived variables increase Daikon's time and memory usage, sometimes dramatically. If false, individual kinds of derived variables can be enabled or disabled individually using configuration options under '`daikon.derive`'. The default value is 'false'.

'`daikon.derive.binary.SequenceFloatIntersection.enabled`'
: Boolean. True iff SequenceFloatIntersection derived variables should be generated. The default value is 'false'.

'`daikon.derive.binary.SequenceFloatSubscript.enabled`'
: Boolean. True iff SequenceFloatSubscript derived variables should be generated. The default value is 'true'.

'`daikon.derive.binary.SequenceFloatSubsequence.enabled`'
: Boolean. True iff SequenceFloatSubsequence derived variables should be generated. The default value is 'false'.

'`daikon.derive.binary.SequenceFloatUnion.enabled`'
: Boolean. True iff SequenceFloatUnion derived variables should be generated. The default value is 'false'.

'`daikon.derive.binary.SequenceScalarIntersection.enabled`'
: Boolean. True iff SequenceScalarIntersection derived variables should be generated. The default value is 'false'.

'daikon.derive.binary.SequenceScalarSubscript.enabled'
Boolean. True iff SequenceScalarSubscript derived variables should be generated. The default value is 'true'.

'daikon.derive.binary.SequenceScalarSubsequence.enabled'
Boolean. True iff SequenceScalarSubsequence derived variables should be generated. The default value is 'false'.

'daikon.derive.binary.SequenceScalarUnion.enabled'
Boolean. True iff SequenceScalarUnion derived variables should be generated. The default value is 'false'.

'daikon.derive.binary.SequenceStringIntersection.enabled'
Boolean. True iff SequenceStringIntersection derived variables should be generated. The default value is 'false'.

'daikon.derive.binary.SequenceStringSubscript.enabled'
Boolean. True iff SequenceStringSubscript derived variables should be generated. The default value is 'true'.

'daikon.derive.binary.SequenceStringSubsequence.enabled'
Boolean. True iff SequenceStringSubsequence derived variables should be generated. The default value is 'false'.

'daikon.derive.binary.SequenceStringUnion.enabled'
Boolean. True iff SequenceStringUnion derived variables should be generated. The default value is 'false'.

'daikon.derive.binary.SequencesConcat.enabled'
Boolean. True iff SequencesConcat derived variables should be created. The default value is 'false'.

'daikon.derive.binary.SequencesJoin.enabled'
Boolean. True iff SequencesJoin derived variables should be generated. The default value is 'false'.

'daikon.derive.binary.SequencesJoinFloat.enabled'
Boolean. True iff SequencesJoin derived variables should be generated. The default value is 'false'.

'daikon.derive.binary.SequencesPredicate.boolOnly'
Boolean. True if Daikon should only generate derivations on boolean predicates. The default value is 'true'.

'daikon.derive.binary.SequencesPredicate.enabled'
Boolean. True iff SequencesPredicate derived variables should be generated. The default value is 'false'.

'daikon.derive.binary.SequencesPredicate.fieldOnly'
Boolean. True if Daikon should only generate derivations on fields of the same data structure. The default value is 'true'.

'daikon.derive.binary.SequencesPredicateFloat.boolOnly'
Boolean. True if Daikon should only generate derivations on boolean predicates. The default value is 'true'.

'`daikon.derive.binary.SequencesPredicateFloat.enabled`'
: Boolean. True iff SequencesPredicate derived variables should be generated. The default value is 'false'.

'`daikon.derive.binary.SequencesPredicateFloat.fieldOnly`'
: Boolean. True if Daikon should only generate derivations on fields of the same data structure. The default value is 'true'.

'`daikon.derive.ternary.SequenceFloatArbitrarySubsequence.enabled`'
: Boolean. True iff SequenceFloatArbitrarySubsequence derived variables should be generated. The default value is 'false'.

'`daikon.derive.ternary.SequenceScalarArbitrarySubsequence.enabled`'
: Boolean. True iff SequenceScalarArbitrarySubsequence derived variables should be generated. The default value is 'false'.

'`daikon.derive.ternary.SequenceStringArbitrarySubsequence.enabled`'
: Boolean. True iff SequenceStringArbitrarySubsequence derived variables should be generated. The default value is 'false'.

'`daikon.derive.unary.SequenceInitial.enabled`'
: Boolean. True iff SequenceInitial derived variables should be generated. The default value is 'false'.

'`daikon.derive.unary.SequenceInitialFloat.enabled`'
: Boolean. True iff SequenceInitial derived variables should be generated. The default value is 'false'.

'`daikon.derive.unary.SequenceLength.enabled`'
: Boolean. True iff SequenceLength derived variables should be generated. The default value is 'true'.

'`daikon.derive.unary.SequenceMax.enabled`'
: Boolean. True iff SequencesMax derived variables should be generated. The default value is 'false'.

'`daikon.derive.unary.SequenceMin.enabled`'
: Boolean. True iff SequenceMin derived variables should be generated. The default value is 'false'.

'`daikon.derive.unary.SequenceSum.enabled`'
: Boolean. True iff SequenceSum derived variables should be generated. The default value is 'false'.

'`daikon.derive.unary.StringLength.enabled`'
: Boolean. True iff StringLength derived variables should be generated. The default value is 'false'.

6.1.1.5 Simplify interface configuration options

The configuration options in this section are used to customize the interface to the Simplify theorem prover. See the description of the '`--suppress_redundant`' command-line option in Section 4.2 [Options to control invariant detection], page 19.

‘`daikon.simplify.LemmaStack.print_contradictions`’
: Boolean. Controls Daikon’s response when inconsistent invariants are discovered while running Simplify. If true, Daikon will print an error message to the standard error stream listing the contradictory invariants. This is mainly intended for debugging Daikon itself, but can sometimes be helpful in tracing down other problems. For more information, see the section on troubleshooting contradictory invariants in the Daikon manual. The default value is ‘false’.

‘`daikon.simplify.LemmaStack.remove_contradictions`’
: Boolean. Controls Daikon’s response when inconsistent invariants are discovered while running Simplify. If false, Daikon will give up on using Simplify for that program point. If true, Daikon will try to find a small subset of the invariants that cause the contradiction and avoid them, to allow processing to continue. For more information, see the section on troubleshooting contradictory invariants in the Daikon manual. The default value is ‘true’.

‘`daikon.simplify.LemmaStack.synchronous_errors`’
: Boolean. If true, ask Simplify to check a simple proposition after each assumption is pushed, providing an opportunity to wait for output from Simplify and potentially receive error messages about the assumption. When false, long sequences of assumptions may be pushed in a row, so that by the time an error message arrives, it’s not clear which input caused the error. Of course, Daikon’s input to Simplify isn’t supposed to cause errors, so this option should only be needed for debugging. The default value is ‘false’.

‘`daikon.simplify.Session.simplify_max_iterations`’
: A non-negative integer, representing the largest number of iterations for which Simplify should be allowed to run on any single conjecture before giving up. Larger values may cause Simplify to run longer, but will increase the number of invariants that can be recognized as redundant. The default value is small enough to keep Simplify from running for more than a few seconds on any one conjecture, allowing it to verify most simple facts without getting bogged down in long searches. A value of 0 means not to bound the number of iterations at all, though see also the `simplify_timeout` parameter.. The default value is ‘1000’.

‘`daikon.simplify.Session.simplify_timeout`’
: A non-negative integer, representing the longest time period (in seconds) Simplify should be allowed to run on any single conjecture before giving up. Larger values may cause Simplify to run longer, but will increase the number of invariants that can be recognized as redundant. Roughly speaking, the time spent in Simplify will be bounded by this value, times the number of invariants generated, though it can be much less. A value of 0 means to not bound Simplify at all by time, though also see the option `simplify_max_iterations`. Beware that using this option might make Daikon’s output depend on the speed of the machine it’s run on. The default value is ‘0’.

‘`daikon.simplify.Session.trace_input`’
: Boolean. If true, the input to the Simplify theorem prover will also be directed to a file named simplifyN.in (where N is a number starting from 0) in the current

directory. Simplify's operation can then be reproduced with a command like '`Simplify -nosc <simplify0.in`'. This is intended primarily for debugging when Simplify fails. The default value is 'false'.

'`daikon.simplify.Session.verbose_progress`'
: Positive values mean to print extra indications as each candidate invariant is passed to Simplify during the `--suppress_redundant` check. If the value is 1 or higher, a hyphen will be printed when each invariant is passed to Simplify, and then replaced by a '`T`' if the invariant was redundant, '`F`' if it was not found to be, and '`?`' if Simplify gave up because of a time limit. If the value is 2 or higher, a '`<`' or '`>`' will also be printed for each invariant that is pushed onto or popped from from Simplify's assumption stack. This option is mainly intended for debugging purposes, but can also provide something to watch when Simplify takes a long time. The default value is '0'.

6.1.1.6 General configuration options

This section lists miscellaneous configuration options for Daikon.

'`daikon.Daikon.calc_possible_invs`'
: Boolean. Just print the total number of possible invariants and exit. The default value is 'false'.

'`daikon.Daikon.disable_splitting`'
: Boolean. Controls whether or not splitting based on the built-in splitting rules is disabled. The built-in rules look for implications based on boolean return values and also when there are exactly two exit points from a method. The default value is 'false'.

'`daikon.Daikon.enable_floats`'
: Boolean. Controls whether invariants are reported over floating-point values. The default value is 'true'.

'`daikon.Daikon.guardNulls`'
: If "always", then invariants are always guarded. If "never", then invariants are never guarded. If "missing", then invariants are guarded only for variables that were missing ("can be missing") in the dtrace (the observed executions).

  Guarding means adding predicates that ensure that variables can be dereferenced. For instance, if `a` can be null — that is, if `a.b` can be nonsensical — then the guarded version of '`a.b == 5`' is '`(a != null) ==> (a.b == 5)`'.

  (To do: Some configuration option (maybe this one) should add guards for other reasons that lead to nonsensical values (see Section 5.3 [Variable names], page 24).) The default value is 'default'.

'`daikon.Daikon.internal_check`'
: When true, perform detailed internal checking. These are essentially additional, possibly costly assert statements. The default value is 'false'.

'`daikon.Daikon.output_conditionals`'
: Boolean. Controls whether conditional program points are displayed. The default value is 'true'.

'`daikon.Daikon.ppt_perc`'
: Integer. Percentage of program points to process. All program points are sorted by name, and all samples for the first `ppt_perc` program points are processed. A percentage of 100 matches all program points. The default value is '100'.

'`daikon.Daikon.print_sample_totals`'
: Boolean. Controls whether or not the total samples read and processed are printed at the end of processing. The default value is 'false'.

'`daikon.Daikon.progress_delay`'
: The amount of time to wait between updates of the progress display, measured in milliseconds. A value of -1 means do not print the progress display at all. The default value is '1000'.

'`daikon.Daikon.progress_display_width`'
: The number of columns of progress information to display. In many Unix shells, this can be set to an appropriate value by '`--config_option daikon.Daikon.progress_display_width=$COLUMNS`'. The default value is '80'.

'`daikon.Daikon.quiet`'
: Boolean. Controls whether or not processing information is printed out. Setting this variable to true also automatically sets `progress_delay` to -1. The default value is 'false'.

'`daikon.Daikon.show_stack_trace`'
: If true, show stack traces for errors such as file format errors. The default value is 'false'.

'`daikon.Daikon.suppressSplitterErrors`'
: When true compilation errors during splitter file generation will not be reported to the user. The default value is 'false'.

'`daikon.Daikon.undo_opts`'
: Boolean. Controls whether the Daikon optimizations (equality sets, suppressions) are undone at the end to create a more complete set of invariants. Output does not include conditional program points, implications, reflexive and partially reflexive invariants. The default value is 'false'.

'`daikon.Daikon.use_dynamic_constant_optimization`'
: Whether to use the dynamic constants optimization. This optimization doesn't instantiate invariants over constant variables (i.e., that that have only seen one value). When the variable receives a second value, invariants are instantiated and are given the sample representing the previous constant value. The default value is 'true'.

'`daikon.Debug.logDetail`'
: Determines whether or not detailed info (such as from `add_modified`) is printed. The default value is 'false'.

'`daikon.Debug.showTraceback`'
: Determines whether or not traceback information is printed for each call to log. The default value is 'false'.

'`daikon.DynamicConstants.OneOf_only`'
: Boolean. If true only create OneOf invariants for variables that are constant for the entire run. If false, all possible invariants are created between constants. Note that setting this to true only fails to create invariants between constants. Invariants between constants and non-constants are created regardless.

  A problem occurs with merging when this is turned on. If a var_info is constant at one child slice, but not constant at the other child slice, interesting invariants may not be merged because they won't exist on the slice with the constant. This is thus currently defaulted to false. The default value is 'false'.

'`daikon.FileIO.add_changed`'
: Boolean. When false, set modbits to 1 iff the printed representation has changed. When true, set modbits to 1 if the printed representation has changed; leave other modbits as is. The default value is 'true'.

'`daikon.FileIO.check_bb_connections`'
: If true, check all of the basic blocks that make up a function to ensure that there is a path from function entry to the block. The default value is 'true'.

'`daikon.FileIO.continue_after_file_exception`'
: Boolean. When true, suppress exceptions related to file reading. This permits Daikon to continue even if there is a malformed trace file. Use this with care: in general, it is better to fix the problem that caused a bad trace file, rather than to suppress the exception. The default value is 'false'.

'`daikon.FileIO.count_lines`'
: Boolean. When false, don't count the number of lines in the dtrace file before reading. This will disable the percentage progress printout. The default value is 'true'.

'`daikon.FileIO.dtrace_line_count`'
: Long integer. If non-zero, this value will be used as the number of lines in (each) dtrace file input for the purposes of the progress display, and the counting of the lines in the file will be suppressed. The default value is '0'.

'`daikon.FileIO.ignore_missing_enter`'
: When true, just ignore exit ppts that don't have a matching enter ppt rather than exiting with an error. Unmatched exits can occur if only a portion of a dtrace file is processed. The default value is 'false'.

'`daikon.FileIO.max_line_number`'
: Integer. Maximum number of lines to read from the dtrace file. If 0, reads the entire file. The default value is '0'.

'`daikon.FileIO.merge_basic_blocks`'
: If true, variables from basic blocks which predominate a basic block X will be included when X is processed. This allows Daikon to find invariants between

variables in different program points (basic blocks in this case). The default value is 'false'.

'`daikon.FileIO.read_samples_only`'
: Boolean. When true, only read the samples, but don't process them. Used to gather timing information. The default value is 'false'.

'`daikon.FileIO.unmatched_procedure_entries_quiet`'
: Boolean. When true, don't print a warning about unmatched procedure entries, which are ignored by Daikon (unless the –nohierarchy switch is provided). The default value is 'false'.

'`daikon.FileIO.verbose_unmatched_procedure_entries`'
: Boolean. If true, prints the unmatched procedure entries verbosely. The default value is 'false'.

'`daikon.PptCombined.asm_path_name`'
: If non-null, we will compute redundant binary variables when creating a CombinedProgramPoint, using the assembly information in the file specified. The default value is 'null'.

'`daikon.PptCombined.rvars_file`'
: If redundant variables are being computed, the results of the redundancy analysis are printed to this stream. See dkconfig_asm_path_name above. The default value is 'null'.

'`daikon.PptRelation.enable_object_user`'
: Boolean. Controls whether the object-user relation is created in the variable hierarchy. The default value is 'false'.

'`daikon.PptSliceEquality.set_per_var`'
: If true, create one equality set for each variable. This has the effect of turning the equality optimization off, without actually removing the sets themselves (which are presumed to exist in many parts of the code). The default value is 'false'.

'`daikon.PptSplitter.dummy_invariant_level`'
: Integer. A value of zero indicates that DummyInvariant objects should not be created. A value of one indicates that dummy invariants should be created only when no suitable condition was found in the regular output. A value of two indicates that dummy invariants should be created for each splitting condition. The default value is '0'.

'`daikon.PptSplitter.split_bi_implications`'
: Split bi-implications into two separate invariants. The default value is 'false'.

'`daikon.PptTopLevel.pairwise_implications`'
: Boolean. If true, create implications for all pairwise combinations of conditions, and all pairwise combinations of exit points. If false, create implications for only the first two conditions, and create implications only if there are exactly two exit points. The default value is 'false'.

'`daikon.PptTopLevel.remove_merged_invs`'
: Remove invariants at lower program points when a matching invariant is created at a higher program point. For experimental purposes only. The default value is 'false'.

'`daikon.PrintInvariants.old_array_names`'
: In the new decl format, print array names without as 'a[]' as opposed to 'a[..]' This creates names that are more compatible with the old output. This option has no effect in the old decl format. The default value is 'true'.

'`daikon.PrintInvariants.print_all`'
: If true, print all invariants without any filtering. The default value is 'false'.

'`daikon.PrintInvariants.print_inv_class`'
: Print invariant classname with invariants in output of `format()` method, normally used only for debugging output rather than ordinary printing of invariants. The default value is 'false'.

'`daikon.PrintInvariants.remove_post_vars`'
: If true, remove as many variables as possible that need to be indicated as 'post'. Post variables occur when the subscript for a derived variable with an orig sequence is not orig. For example: orig(a[post(i)]) An equivalent expression involving only orig variables is substitued for the post variable when one exists. The default value is 'false'.

'`daikon.PrintInvariants.replace_prestate`'
: This option must be given with "–format Java" option.

  Instead of outputting prestate expressions as "\old(E)" within an invariant, output a variable names (e.g. 'v1'). At the end of each program point, output the list of variable-to-expression mappings. For example: with this option set to false, a program point might print like this:

  <pre> foo.bar.Bar(int):::EXIT \old(capacity) == sizeof(this.theArray) </pre>

  With the option set to true, it would print like this:

  <pre> foo.bar.Bar(int):::EXIT v0 == sizeof(this.theArray) prestate assignment: v0=capacity </pre> The default value is 'true'.

'`daikon.PrintInvariants.static_const_infer`'
: This enables a different way of treating static constant variables. They are not created into invariants into slices. Instead, they are examined during print time. If a unary invariant contains a value which matches the value of a static constant varible, the value will be replaced by the name of the variable, "if it makes sense". For example, if there is a static constant variable a = 1. And there exists an invariant x <= 1, x <= a would be the result printed. The default value is 'false'.

'`daikon.PrintInvariants.true_inv_cnt`'
: If true, print the total number of true invariants. This includes invariants that are redundant and would normally not be printed or even created due to optimizations. The default value is 'false'.

'`daikon.ProglangType.convert_to_signed`'
: If true, treat 32 bit values whose high bit is on, as a negative number (rather than as a 32 bit unsigned). The default value is 'false'.

'`daikon.VarInfo.constant_fields_simplify`'
: If true, the treat static constants (such as MapQuick.GeoPoint.FACTOR) as fields within an object rather than as a single name. Not correct, but used to obtain compatibility with VarInfoName. The default value is 'true'.

'`daikon.VarInfo.declared_type_comparability`'
: If true, then variables are only considered comparable if they are declared with the same type. For example, java.util.List is not comparable to java.util.ArrayList and float is not comparable to double. This may miss valid invariants, but significant time can be saved and many variables with different declared types are not comparable (e.g., java.util.Date and java.util.ArrayList). The default value is 'true'.

'`daikon.chicory.DaikonVariableInfo.constant_infer`'
: Enable experimental techniques on static constants. The default value is 'false'.

'`daikon.split.ContextSplitterFactory.granularity`'
: Enumeration (integer). Specifies the granularity to use for callsite splitter processing. 0 is line-level granularity; 1 is method-level granularity; 2 is class-level granularity. The default value is '1'.

'`daikon.split.SplitterFactory.compile_timeout`'
: Positive integer. Specifies the Splitter compilation timeout, in seconds, after which the compilation process is terminated and retried, on the assumption that it has hung. The default value is '6'.

'`daikon.split.SplitterFactory.compiler`'
: String. Specifies which Java compiler is used to compile Splitters. This can be the full path name or whatever is used on the commandline. The default value is 'javac'.

'`daikon.split.SplitterFactory.delete_splitters_on_exit`'
: Boolean. Specifies whether or not the temporary Splitter files should be deleted on exit. The default value is 'true'.

'`daikon.split.SplitterList.all_splitters`'
: Boolean. Enables indiscriminate splitting (see Daikon manual, Section 6.2.2 [Indiscriminate splitting], page 81, for an explanation of this technique). The default value is 'true'.

'`daikon.suppress.NIS.enabled`'
: Boolean. If true, enable non-instantiating suppressions. The default value is 'true'.

'`daikon.suppress.NIS.hybrid_threshhold`'
: Int. Less and equal to this number means use the falsified method in the hybrid method of processing falsified invariants, while greater than this number means use the antecedent method. Empirical data shows that number should not be more than 10000. The default value is '2500'.

'`daikon.suppress.NIS.skip_hashcode_type`'
: Boolean. If true, skip variables of file rep type hashcode when creating invariants over constants in the antecedent method. The default value is 'true'.

'`daikon.suppress.NIS.suppression_processor`'
: Specifies the algorithm that NIS uses to process suppressions. Possible selections are 'HYBRID', 'ANTECEDENT', and 'FALSIFIED'. The default is the hybrid algorithm which uses the falsified algorithm when only a small number of suppressions need to be processed and the antecedent algorithm when a large number of suppressions are processed. The default value is 'HYBRID'.

'`daikon.suppress.NIS.suppressor_list`'
: Boolean. If true, use the specific list of suppressor related invariant prototypes when creating constant invariants in the antecedent method. The default value is 'true'.

6.2 Conditional invariants and implications

Conditional invariants are invariants that are true only part of the time. For instance, the postcondition for the absolute value procedure is

```
if arg < 0
  then return == -arg
  else return == arg
```

The invariant `return == -arg` is a conditional invariant because it depends on the predicate `arg < 0` being true. An *implication* is a compound invariant that includes both the predicate and the conditional invariant (also called the consequent).

Another type of implication is a *context-sensitive* invariant — a fact about method A that is true only when A is called by method B, but not true in general about A. Implications can be used to construct context-sensitive invariants: set a variable that depends on the call site, then compute an implication whose predicate tests that variable. For an example, see the paper "Selecting, refining, and evaluating predicates for program analysis" (`http://groups.csail.mit.edu/pag/daikon/pubs/predicates-tr914-abstract.html`).

Daikon must be supplied with the predicate for an implication. Daikon has certain built-in predicates that it uses for finding conditional invariants; examples are which return statement was executed in a procedure and whether a boolean procedure returns true or false. Additionally, Daikon can read predicates from a file called a splitter info ('`.spinfo`') file and find implications based on those predicates. The splitter info file can be produced automatically, such as by static analysis of the program using the CreateSpinfo and CreateSpinfoC programs or by cluster analysis of the traced values in the data trace file. Details of these techniques and usage guides can be found in Section 6.3 [Enhancing conditional invariant detection], page 83. Users can also create splitter info files by hand or manually augment automatically-created ones.

To detect conditional invariants and implications:

1. Create the splitter info file, either automatically or by hand.
2. Run Daikon with the '`.spinfo`' file as one of its arguments. (The order of arguments does not matter.) For example,

   ```
   java daikon.Daikon Foo.decls Foo.spinfo Foo.dtrace
   ```

The term "splitter" comes from Daikon's technique for detecting implications and conditional invariants. For each predicate, Daikon creates two conditional program points — one for program executions that satisfy the condition and one for those that don't — and splits the data trace into two parts. Invariant detection is then performed on the conditional program points (that is, the parts of the data trace) separately and any invariants detected are reported as conditional invariants (as implications).

To be precise, we say that an invariant holds exclusively if it is discovered on one side of a split, and its negation is discovered on the opposite side. Daikon creates conditional invariants whose predicates are invariants that hold exclusively on one side of a split, and whose consequents are invariants that hold on that side of the split but not on the un-split program point. If Daikon finds multiple exclusive conditions, it will create biconditional ("if and only if") invariants between the equivalent conditions. Within the context of the program, each of the exclusive conditions is equivalent to the splitting condition. In particular, if both the splitting condition and its negation are within the grammar of invariants that Daikon detects, the splitting condition may appear as the predicate of the generated conditional invariants. On the other hand, if other equivalent conditions are found, or if the splitting condition is not expressible in Daikon's grammar, it might not appear in the generated implications.

In some cases, the default policy of selecting predicates from Daikon's output may be insufficient. For instance, Daikon might not detect any invariant equivalent to the splitting condition, if it is sufficiently complex or application-specific. In such situations, Daikon can also use the splitting condition itself as the predicate of an implication, as what is called a "dummy invariant". A "dummy invariant" is one whose meaning is not dealt with directly by Daikon; instead Daikon knows only how to print the invariant in its output. When a tool or a user writes a splitter info file, the file can specify a way to print the condition represented by the splitter in any of Daikon's output formats. If the use of dummy invariants is enabled, invariants with the supplied output formats will be used as the predicates of conditional invariants.

To use dummy invariants, a condition's formatting must be specified in the splitter info file, and the configuration option '`daikon.PptTopLevel.dummy_invariant_level`' must be set to a non-zero value (see Section 6.1.1 [List of configuration options], page 52).

6.2.1 Splitter info file

A splitter info file contains the conditions that Daikon should use to create conditional invariants. Each section in the '`.spinfo`' file consists of a sequence of non-blank lines; sections are separated by blank lines. There are two types of sections: program point sections and replacement sections. See Section 6.2.3 [Example splitter info file], page 82, for an example splitter info file.

6.2.1.1 Program point sections

Program point sections have a line specifying a program point name followed by lines specifying the condition(s) associated with that program point, each condition on its own line. Additional information about a condition may be specified on indented lines. For example, a typical entry is

```
PPT_NAME pptname
condition1
```

```
condition2
    DAIKON_FORMAT output string
    ESC_FORMAT output string
condition3
...
```

pptname can be any string that matches a part of the desired program point name as printed in the decls file. In finding matching program points, Daikon uses the first program point that matches *pptname*. Caution is necessary when dealing with method names that are prefixes of other method names. For instance, if the class `List` has methods `add` and `addAll`, specifying '`PPT_NAME List.add`' might select either method, depending on which was encountered first. Instead writing '`PPT_NAME List.add(`' will match only the `add` method.

Each condition is a Java expression of boolean type. All variables that appear in the condition must also appear in the declaration of the program point in the '`.decls`' file. (In other words, all the variables must be in scope at the program point(s) where the Splitter is intended to operate.) The automatically generated Splitter source code fails to compile (but Daikon proceeds without it) if a variable name in a condition is not found at the matching program point.

Indented lines beginning with '`DAIKON_FORMAT`', '`JAVA_FORMAT`', '`ESC_FORMAT`', or '`SIMPLIFY_FORMAT`' may be used to specify how to represent the condition specified by the splitter in each of Daikon's output formats, to allow the splitting condition to be used as a dummy invariant.

6.2.1.2 Replacement sections

Ordinarily, a splitting condition may not invoke user-defined methods, because when Daikon reads data trace files, it does not have access to the program source. A replace section of the splitter info file can specify the bodies of methods, permitting conditions to invoke those methods. The format is as follows:

```
REPLACE
procedure1
replacement1
procedure2
replacement2
...
```

where *replacementi* is a Java expression for the body of *procedurei*. In each condition, Daikon replaces procedure calls by their replacements. A replace section may appear anywhere in the splitter info file.

6.2.2 Indiscriminate splitting

Ordinarily, each condition in an '`.spinfo`' file gives rise to conditional invariants only at the program point in whose section the condition appears. Alternately, every condition can be used at every program point, regardless of where in the '`.spinfo`' file the condition appeared; this latter approach is called "indiscriminate splitting".

The advantage of indiscriminate splitting is that a condition that is useful at one program point may also be useful at another — if the same variables are in scope or other variables of the same name are in scope. The disadvantage of indiscriminate splitting is that often the condition is not applicable everywhere, and when it is, it may not be useful at all such locations, so checking for many conditional invariants may slow down Daikon without a corresponding benefit. Indiscriminate splitting can result in Daikon attempting to use many conditions that are inappropriate at certain program points, for instance because the program point does not have (in scope) all the variables that are used in the condition. For example, the condition `myArray.length == x` is inapplicable at a program point if either of `myArray` and `x` is not in scope at that program point. In this case, Daikon prints a warning message and proceeds, using conditions wherever they are valid.

By default, Daikon uses indiscriminate splitting. To use non-indiscriminate splitting, place the following line in a file that is passed to Daikon via the '`--config`' flag (see Section 4.4 [Daikon configuration options], page 20):

```
daikon.split.SplitterList.all_splitters = false
```

6.2.3 Example splitter info file

Below is an implementation of a simple Queue for positive integers and a corresponding '`.spinfo`' file. The splitter info file is like the one that `CreateSpinfo` would create for that class, but also demonstrates some other features.

6.2.3.1 Example class

```
class simpleStack {

  private int[] myArray;
  private int currentSize;

  public simpleStack(int capacity) {
    myArray = new int[capacity];
    currentSize = 0;
  }

  /** Adds an element to the back of the stack, if the stack is
    * not full.
    * Returns true if this succeeds, false otherwise.  **/
  public String push(int x) {
    if ( !isFull() && x >= 0) {
      myArray[currentSize] = x;
      currentSize++;
      return true;
    } else {
      return false;
    }
  }

  /** Returns the most recently inserted stack element.
    * Returns -1 if the stack is empty. **/
  public int pop() {
    if ( !isEmpty() ) {
      currentSize--;
      return myArray[currentSize];
    } else {
      return -1;
    }
  }
```

```
 /** Returns true if the stack is empty, false otherwise. **/
  private boolean isEmpty() {
    return (currentSize == 0);
  }

  /** Returns true if the stack is full, false otherwise. **/
  private boolean isFull() {
    return (currentSize == myArray.length);
  }
}
```

6.2.3.2 Resulting .spinfo file

```
REPLACE
isFull()
currentSize == myArray.length
isEmpty()
currentSize == 0

PPT_NAME  simpleStack.push
!isFull() && x >= 0
    DAIKON_FORMAT !isFull() and x >= 0
    SIMPLIFY_FORMAT (AND (NOT (isFull this)) (>= x 0))

PPT_NAME  simpleStack.pop
!isEmpty()

PPT_NAME  simpleStack.isFull
currentSize == myArray.length - 1

PPT_NAME  simpleStack.isEmpty
currentSize == 0
```

6.3 Enhancing conditional invariant detection

The built-in mechanisms (see Section 6.2 [Conditional invariants], page 79) have limitations in the invariants they can find. By supplying splitting conditions to Daikon via a splitter info file, the user can infer more conditional invariants. To ease this task, there are two methods to automatically create splitter info files for use by Daikon.

6.3.1 Static analysis for splitters

In static analysis, all explicitly stated boolean statements in the program source are extracted and used as splitting conditions. The assumption is that conditions that are explicitly tested in the program are likely to affect the program's behavior and could lead to useful conditional invariants.

The CreateSpinfo program takes Java source code as input and creates a splitter info file for each input file; for instance,

```
java daikon.tools.jtb.CreateSpinfo Foo.java Bar.java
```

creates the splitter info files '`Foo.spinfo`' and '`Bar.spinfo`'. Given an '`-o` *`filename`*' argument, CreateSpinfo puts all the splitters in the specified file instead. The resulting splitter info file(s) contains each boolean expression that appears in the source code. The conditional statements that the programmer used in the source code are likely to have important semantic properties. This simple heuristic of using these conditional statements as predicates for conditional invariant detection is often quite effective.

If you get an error such as

```
jtb.ParseException: Encountered ";" at line 253, column 8.
Was expecting one of: "abstract" ...
```

then you may have encountered a bug in the JTB library on which CreateSpinfo is built. It does not permit empty declarations in a class body. Remove the extra semicolon in your Java file (at the indicated position) and re-rerun CreateSpinfo.

The `CreateSpinfoC` program performs the same function, for C source code. It can only be run on postprocessed source files—that is, source files contain no CPP commands. CPP commands are lines starting with '`#`', such as '`#include`'. To expand CPP commands into legal C, run either `cpp -P` or `gcc -P -E`. For instance, here is how you could use it.

```
cpp -P foo.c foo.c-expanded
cpp -P bar.c bar.c-expanded
java daikon.tools.jtb.CreateSpinfoC \
     foo.c-expanded bar.c-expanded
```

If you get an error such as

```
... Lexical error at line 5, column 1.
Encountered: "#" (35), after : ""
```

then you forgot to run CPP before running CreateSpinfoC.

6.3.2 Cluster analysis for splitters

Cluster analysis is a statistical method that finds groups or clusters in data. The clusters may indicate conditional properties in the program. A conditional property at a program point separates the data into those that satisfy it and those that do not, and conditional invariants can be induced by clustering. Any invariant that is discovered over one cluster but not over another is a conditional invariant—the predicate for the conditional invariant being membership in the cluster in which the invariant was found. The cluster analysis mechanism finds clusters in the data trace file, infers invariants over any clusters that it finds, and writes these invariants into a splitter info file for conditional invariant detection.

To find splitting conditions using cluster analysis, run the `runcluster.pl` program (found in the '`$DAIKONDIR/bin`' directory) in the following way:

```
runcluster.pl [options] dtrace_file ... decls_files ...
```

The *options* are:

'`-a` *ALG*'
'`--algorithm` *ALG*'
: *ALG* specifies a clustering algorithm. Current options are '`km`' (for kmeans), '`hierarchical`', and '`xm`' (for xmeans). The default is '`xm`'.

'`-k`'
: The number of clusters to use (for algorithms which require this input, which is everything except xmeans). The default is 4.

'`--keep`'
: Don't delete the temporary files created by the clustering process. This is a debugging flag.

The `runcluster.pl` script currently supports three clustering programs. They are implementations of the kmeans algorithm, hierarchical clustering and the xmeans algorithm (kmeans algorithm with efficient discovery of the number of clusters). The kmeans and hierarchical clustering tools are provided in the Daikon distribution. The xmeans code and executable are publicly available at `http://www.cs.cmu.edu/~dpelleg/kmeans.html` (fill in the license form and mail it in).

6.3.3 Random selection for splitters

Random selection can create representative samples of a data set with the added benefit of finding conditional properties and eliminating outliers. Given traced data, the TraceSelect tool creates several small subsets of the data by randomly selecting parts of the original trace file. Any invariant that is discovered in the smaller samples but not found over the entire data is a conditional invariant.

To find splitting conditions using random selection, run the `daikon.tools.TraceSelect` program in the following way:

```
java daikon.tools.TraceSelect num_reps sample_size [options] \
    dtrace_file decls_files ... [daikon_options]
```

num_reps is the number of subsets to create, and *sample_size* is the number of invocations to collect for each method.

The *daikon_options* are the same options that can be provided to the daikon.Daikon program.

The *options* for TraceSelect are:

'`-NOCLEAN`'
: Don't delete the temporary trace samples created by the random selection process. This can help for debugging or for using the tool solely to create trace samples instead of calculating invariants over the samples.

'`-INCLUDE_UNRETURNED`'
: Allows random selection to choose method invocations that entered the method successfully but did not exit normally; either from a thrown Exception or abnormal termination.

'`-DO_DIFFS`'
: Creates an spinfo file for generating conditional invariants and implications by reporting the invariants that appear in at least one of the samples but not over the entire data set.

6.4 Dynamic abstract type inference (DynComp)

Abstract types group variables that are used for related purposes in a program. For example, suppose that some `int` variables in your program are array lengths or indices, and other `int` variables represent time. Even thought these variables have the same type (`int`) in the programming language, they have different abstract types.

Abstract types can be provided as additional input to Daikon, so that it only infers invariants between values of the same abstract type. This can improve Daikon's performance, because it reduces the number of potential invariants that must be checked, and also improve the relevance of its output, since invariants over unrelated variables are superfluous for many tasks. The Daikon distribution includes a tool named DynComp that dynamically infers abstract types (also called comparability types) from program executions. (In fact, there are two implementations of DynComp that use the same algorithm, one for Java programs and one for binaries compiled from C and C++ source code. When confusion would otherwise arise, we distinguish them as DynCompJ (or DynComp for Java) and DynCompB (or Dyncomp for C/C++) respectively.)

Because abstract type inference must be performed before Daikon runs, it is integrated with the front-ends rather than directly as part of Daikon.

- The Java DynComp tool produces a comparability file that must then be supplied to the Chicory Java front-end. For examples of using DynComp with Java programs, see Section 3.1.2 [Using DynComp with Java programs], page 8. For full details about the DynComp tool for Java, see Section 7.2 [DynComp for Java], page 94.
- The Kvasir front-end for C/C++ binaries has a DynComp mode in which it produces a separate '`.decls`' file containing comparability information, which must be supplied to Daikon along with the '`.dtrace`' file. For examples of using DynComp with C programs, see Section 3.2.2 [Using DynComp with C programs], page 12. For full details about the DynComp tool for C/C++, see Section 7.3.3 [DynComp for C/C++], page 107.

6.5 Loop invariants

Daikon does not by default output loop invariants. Daikon can detect invariants at any location where it is provided with variable values, but currently Daikon's front ends do not supply Daikon with variable values at loop heads.

You could extend a front end to output more variable values, or you could write a new front end.

Alternately, here is a way to use the current front ends to produce loop invariants. This workaround requires you to change your program, but it requires no change to Daikon or its front ends.

At the top of a loop (or at any other location in the program at which you would like to obtain invariants), insert a call to a dummy procedure that does not work but returns immediately. Pass, as arguments to the dummy procedure, all variables of interest (including local variables). Daikon will produce (identical) preconditions and postconditions for the dummy procedure; these are properties that held at the call site.

For instance, you might change the original code

```
public void calculate(int x) {
  int tmp = 0;
  while (x > 0) {
    // you desire to compute an invariant here
    tmp=tmp+x;
    x=x-1;
  }
}
```

into

```
public void calculate(int x) {
```

```
  int tmp = 0;
  while (x > 0) {
    calculate_loophead(x, tmp);
    tmp=tmp+x;
    x=x-1;
  }
}

// dummy procedure
public void calculate_loophead(int x, int tmp) {
}
```

7 Front ends (instrumentation)

The Daikon invariant detector is a machine learning tool that finds patterns (invariants) in data. That data can come from any source, but Daikon is typically used to find invariants over variable values in running programs. A front end is a tool that converts data from some other format into Daikon's input format. The most common type of front end is an instrumenter, which causes your program to output a '`.dtrace`' file that Daikon can process.

This chapter describes several front ends (instrumenters) that are part of Daikon. It is relatively easy to build your own front end, if these do not serve your purpose; we are aware of a number of users who have done so. For more information about building a new front end, see section "New front ends" in *Daikon Developer Manual*.

7.1 Java front end Chicory

The Daikon front end for Java, named Chicory, executes Java programs, creates data trace ('`.dtrace`') files, and optionally runs Daikon on them. Chicory is named after the chicory plant, whose root is sometimes used as a coffee substitute or flavor enhancer.

To use Chicory, run your program as you normally would, but replace the `java` command with `java daikon.Chicory`. For instance, if you usually run

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

then instead you would run

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

This runs your program and creates file '`MyClass.dtrace`' in the current directory. Furthermore, a single command can both create a trace file and run Daikon:

```
java daikon.Chicory --daikon mypackage.MyClass arg1 arg2 arg3
```

See below for more options.

That's all there is to it! Since Chicory instruments class files directly as they are loaded into Java, you do not need to perform separate instrumentation and recompilation steps. However, you should compile your program with debugging information enabled (the '`-g`' command-line switch to `javac`); otherwise, Chicory uses the names `arg0`, `arg1`, ... as the names of method arguments.

Chicory must be run in a version 5.0 JVM, but it is backward-compatible with older versions of Java code.

7.1.1 Chicory options

Chicory is invoked as follows:

```
java daikon.Chicory chicory-args classname args
```

where

```
java classname args
```

is a valid invocation of Java.

This section lists the optional command-line arguments to Chicory, which appear before the *classname* on the Chicory command line.

7.1.1.1 Program points in Chicory output

This section lists options that control which program points appear in Chicory's output.

'`--ppt-select-pattern=`*regexp*'

Only produce trace output for classes/procedures/program points whose names match the given regular expression. This option may be supplied multiple times, and may be used in conjunction with '`--ppt-omit-pattern`'.

When this switch is supplied, filtering occurs in the following way: for each program point, Chicory checks the fully qualified class name, the method name, and the the program point name against each *regexp* that was supplied. If any of these match, then the program point is included in the instrumentation.

Suppose that method `bar` is defined only in class `C`. Then to traces only `bar`, you could match the method name (in any class) with regular expression '`bar$`', or you could match the program point name with '`C\.bar\(`'.

Using Unix shell syntax, you would execute

```
java daikon.Chicory --ppt-select-pattern='bar$' ...
java daikon.Chicory --ppt-select-pattern='C\.bar\(' ...
```

From the Windows command line, you would execute

```
java daikon.Chicory --ppt-select-pattern='bar$' ...
java daikon.Chicory --ppt-select-pattern='C\.bar\(' ...
```

'`--ppt-omit-pattern=`*regexp*'

Do not produce data trace output for classes/procedures/program points whose names match the given regular expression. This reduces the size of the data trace file and also may make the instrumented program run faster, since it need not output those variables.

This option works just like '`--ppt-select-pattern`' does, except that matching program points are excluded, not included.

The '`--ppt-omit-pattern`' argument may be supplied multiple times, in order to specify multiple omitting criteria. A program point is omitted if its fully qualified class, procedure name, or complete program point name matches one of the omitting criteria.

The '`--ppt-omit-pattern`' argument may be supplied multiple times, in order to specify multiple omitting criteria. A program point is omitted if its fully qualified class, fully qualified procedure name, or complete program point name exactly matches one of the omitting criteria. A regular expression matches if it matches any portion of the program point name. Note that currently only classes are matched, not each full program point name. Thus, either all of a class's methods are traced, or none of them are.

Here are examples of how to avoid detecting invariants over various parts of your program.

- omit a whole package:

```
java daikon.Chicory '--ppt-omit-pattern=^junit\.'
java daikon.Chicory '--ppt-omit-pattern=^daikon\.util\..*' ...
```

- omit a single class:

```
java daikon.Chicory '--ppt-omit-pattern=HashSetLinear\$HslIterator' ...
```

- omit a single method:

```
java daikon.Chicory '--ppt-omit-pattern=StackAr.topAndPop()' \
     ...
```

- omit a single program point:

```
java daikon.Chicory '--ppt-omit-pattern=StackAr.<init>(int):::EXIT33' ..
```

7.1.1.2 Variables in Chicory output

This section lists options that control which variables appear in Chicory's output.

'`--nesting-depth=n`'

Depth to which to examine structure components (default 2). This parameter determines which variables the front end causes to be output at runtime. For instance, suppose that a program contained the following data structures and variables:

```
class A {
  int x;
  B b;
}
class B {
  int y;
  int z;
}
A myA;

class Link {
  int val;
  Link next;
}
Link myList;
```

- If depth=0, only the identities (hashcodes) of `myA` and `myList` would be examined; those variables could be determined to be equal or not equal to other variables.
- If depth=1, then also `MyA.b`, `myList.next`, and the integers `myA.x` and `myList.val` would be examined.
- If depth=2, then also `MyA.b.y`, `MyA.b.y`, `myList.next.next`, and `myList.next.val` would be examined.

Values whose value is undefined are not examined. For instance, if `myA` is `null` on a particular execution of a program point, then `myA.b` is not accessed on that execution regardless of the depth parameter. That variable appears in the '`.dtrace`' file, but its value is marked as nonsensical.

'`--omit-var=regex`'

Do not include variables whose name matches the regular expression. Variables will be omitted from each program point in which they appear.

'`--std-visibility`'

When this switch is on, Chicory will traverse exactly those fields that are visible from a given program point. For instance, only the public fields of class `pack1.B` will be included at a program point for class `pack2.A` whether or not `pack1.B` is instrumented. By default, Chicory outputs all fields in instrumented classes (even those that would not be accessible in Java code at the given program point) and outputs no fields from uninstrumented classes (even those that are accessible). When you supply '`--std-visibility`', consider also supplying '`--purity-file`' to enrich the set of expressions in Daikon's output.

'`--purity-file=`*`pure-methods-file`*'

File *pure-methods-file* lists the pure methods (sometimes called observer methods) in a Java program. Pure methods have no externally side effects, such as setting variables or producing output. For example, most implementations of the `hashCode()`, `toString()`, and `equals()` methods are pure.

For each variable, Chicory adds to the trace new "fields" that represent invoking each pure method on the variable. (Currently, Chicory does so only for pure methods that take no parameters, and obviously this mechanism is only useful for methods that return a value: a pure method that returns no value does nothing!)

Here is an example:

```
class Point {
  private int x, y;
  public int radiusSquared() {
    return x*x + y*y;
  }
}
```

I `radiusSquared()` has been specified as pure, then for each point *p*, Chicory will output the variables *`p`*`.x`, *`p`*`.y`, and *`p`*`.radiusSquared()`. Use of pure methods can improve the Daikon output, since they represent information that the programmer considered important but that is not necessarily stored in a variable.

Invoking a pure method at any time in an application should not change the application's behavior. If a non-pure method is listed in a purity file, then application behavior can change. Chicory does not verify the purity of methods listed in the purity file.

The purity file lists a set of methods, one per line. The format of each method is given by the Sun JDK API:

> The string is formatted as the method access modifiers, if any, followed by the method return type, followed by a space, followed by the class declaring the method, followed by a period, followed by the method name, followed by a parenthesized, comma-separated list of the method's formal parameter types. If the method throws checked exceptions, the parameter list is followed by a space, followed by the word throws followed by a comma-separated list of the thrown exception types. For example:

```
public boolean java.lang.Object.equals(java.lang.Object)
```

The access modifiers are placed in canonical order as specified by "The Java Language Specification". This is public, protected or private first, and then other modifiers in the following order: abstract, static, final, synchronized native.

By convention, *pure-methods-file* has the suffix '`.pure`'. If *pure-methods-file* is specified as a relative (not absolute) file name, it is searched for in the configuration directory specified via '`--configs=`*`directory`*', or in the current directory if no configuration directory is specified.

One way to create a '`.pure`' file is to run the Purity Analysis Kit (`http://jppa.sourceforge.net/`). If you supply the '`--daikon-purity-file`' when running the Purity Analysis Kit, it writes a file that can be supplied to Daikon.

7.1.1.3 Chicory miscellaneous options

This section lists all other Chicory options — that is, all options that do not control which program points and variables appear in Chicory's output.

'`--help`' Print a help message.

'`--debug`' Produce debugging information.

'`--default_bcel`'
: Chicory uses the Byte Code Engineering Library (BCEL) to instrument classfiles. Errors can occur if the application uses an incompatible version of BCEL. By default, Chicory identifies and loads its copy of BCEL when multiple copies of BCEL are in the classpath. It will also issue a warning if multiple copies of BCEL are in the classpath and the application version is not the first one. When this option is chosen, Chicory will simply use whatever version of BCEL is found on the classpath.

'`--sample-start=`*`sample-cnt`*'
: When this option is chosen, Chicory will record each program point until that program point has been executed *sample-cnt* times. Chicory will then begin sampling. Sampling starts at 10% and decreases by a factor of 10 each time another sample-cnt samples have been recorded. If sample_start is 0, then all calls will be recorded.

'`--dtrace-file=`*`filename`*'
: Specifies the default name for the trace output ('`.dtrace`') file. If this is not specified, then the value of the `DTRACEFILE` environment variable (at the time the instrumented program runs) is used. If that environment variable is not used, then the default is '`./`*`CLASSNAME`*`.dtrace`'.

  If the `DTRACEAPPEND` environment variable is set to any value, the dtrace file will be appended to instead of overwritten. Compressed data trace files may not be appended to. In some cases you may find a single large data trace file more convenient; in other cases, a collection of smaller data trace files may give you more control over which subsets of runs to invoke Daikon on.

'`--comparability-file=`*`filename`*'
: This option specifies a declaration file (see section "Declarations" in *Daikon Developer Manual*) that contains comparability information. This information will be incorporated in the output of Chicory. Any variables not included in the comparability file will have their comparability set so that they are comparable to all other variables of the same type.

'`--output-dir=`*`directory`*'
: Write the '`.dtrace`' trace output file to the specified directory. The default is the current directory.

'`--config-dir=`*`directory`*'
: Chicory will use this location to search for configuration files. Currently, this only includes '`*.pure`' files.

'`--daikon`'
: After creating a data trace ('`.dtrace`') file, run Daikon on it. To specify arguments to Daikon use the '`--daikon-args`' option. Also see the '`--daikon-online`' option.

  This option supplies Daikon with a single trace from one execution of your program. By contrast to this option (and '`--daikon-online`'), if you invoke Daikon from the command line, you can supply Daikon with as many trace files as you wish.

'`--daikon-online`'
: This option is like '`--daikon`', except that no '`.dtrace`' data trace file is produced. Instead, Chicory sends trace information over a socket to Daikon, which processes the information incrementally ("online"), as Chicory produces it.

  Just like with the '`--daikon`' option, Daikon is only given a single trace from one execution of your program.

'`--daikon-args=`*`arguments`*'
: Specifies arguments to be passed to Daikon if the –daikon or –daikon-online options are used.

'`--premain=`*`path`*'
: Specifies the absolute pathname to the '`ChicoryPremain.jar`' file. Chicory requires this jar file in order to execute. By default Chicory looks for the jar file in the classpath and in '`$(DAIKONDIR)/java`' (where `DAIKONDIR` is an environment variable that points to the complete installation of Daikon).

  Chicory can also use the '`daikon.jar`' file for this purpose. If it doesn't find '`ChicoryPremain.jar`' above, it will use '`daikon.jar`' itself (if a file named '`daikon.jar`' appears in the classpath). If the Daikon jar file is not named '`daikon.jar`', you can use this switch to specify its name. For example:

  ```
  --premain=C:\lib\daikon-4.1.3.jar
  ```

'`--heap-size=`*`max_heap`*'
: Specifies the maximum size, in bytes, of the memory allocation pool for the target program. The size is specified in the same manner as the -Xmx switch to java.

7.1.2 Static fields (global variables)

Chicory (Daikon's front end for Java) outputs the values of static fields in the current class, but not in other classes. That means that Daikon cannot report properties over static fields in other classes, because it never sees their values. (By contrast, Kvasir (see Section 7.3 [Kvasir], page 100) supplies the values of C/C++ global variables to Daikon.)

If you need Daikon to include all static variables when processing each class, then ask the maintainers to add that feature to Chicory (or work with them to implement the enhancement). In the meanwhile, here are two workarounds.

1. Add a static field whose type is the class containing the fields of interest. You don't have to ever assign to the new field. A disadvantage of this approach is that it gives you properties over the global variables as observed by each class (which might be different).
2. At the beginning and end of each method, add a call to a dummy method that has access to all the globals (via adding the field mentioned above). This produces a single formula that is valid for all global variables at all times.

7.2 DynComp dynamic comparability (abstract type) analysis for Java

The DynComp dynamic comparability analysis tool performs dynamic type inference to group variables at each program point into comparability sets (see section "Program point declarations" in *Daikon Developer Manual* for the numeric representation format of these sets). All variables in each comparability set belong to the same "abstract type" of data that the programmer likely intended to represent, which is a richer set of types than the few basic declared types (e.g., int, float) provided by the language. Consider the example below:

```
public class Year {
    public static void main(String[] args) {
        int year = 2005;
        int winterDays = 58;
        int summerDays = 307;
        compute(year, winterDays, summerDays);
    }

    public static int compute(int yr, int d1, int d2) {
        if (0 != yr % 4)
            return d1 + d2;
        else
            return d1 + d2 + 1;
    }
}
```

The three variables in `main()` all have the same Java representation type, `int`, but two of them hold related quantities (numbers of days), as can be determined by the fact that they interact when the program adds them, whereas the other contains a conceptually distinct quantity (a year). The abstract types 'day' and 'year' are both represented as `int`, but DynComp can differentiate them with its dynamic analysis. For example, DynComp can infer that `winterDays` and `summerDays` are comparable (belong to the same abstract type) because the program adds their values together within the compute() function.

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

To use DynComp, run your program as you normally would, but replace the `java` command with `java daikon.DynComp`. For instance, if you usually run

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

then instead you would run

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

This runs your program and creates the file '`MyClass.decls-DynComp`' in the current directory. DynComp also creates '`MyClass.txt-cset`', which contains the same information and a further level of detail in an easier-to-read format. The '`.decls`' file may be passed to Chicory, as described in Section 3.1.2 [Using DynComp with Java programs], page 8.

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

See below for more options.

While you may run DynComp with the standard JDK, using the '`--no-jdk`' switch, you can obtain more accurate results by using a copy of the JDK that has been instrumented with DynComp. See Section 7.2.1 [Instrumenting the JDK with DynComp], page 97, below, for instructions.

This is part of a sample '`.decls`' file generated by running DynComp on the example above:

```
DECLARE
Year.compute(int, int, int):::ENTER
yr
int # isParam=true
int
3
d1
int # isParam=true
int
2
d2
int # isParam=true
int
2

DECLARE
Year.compute(int, int, int):::EXIT11
```

```
yr
int # isParam=true
int
3
d1
int # isParam=true
int
2
d2
int # isParam=true
int
2
return
int
int
2
```

The declarations file format is described in section "Program point declarations" in *Daikon Developer Manual*.

DynComp creates two representations of the comparability information in the files '`foo.txt-cset`' and '`foo.txt-trace`'. In the '`cset`' file, DynComp outputs comparability sets as sets. The above '`.decls`' output corresponds to the following in '`cset`':

```
Daikon Variable sets for Year.compute(I yr, I d1, I d2) enter
  [1] [daikon.chicory.ParameterInfo:d1]
  [1] [daikon.chicory.ParameterInfo:d2]
  [1] [daikon.chicory.ParameterInfo:yr]
Daikon Variable sets for Year.compute(I yr, I d1, I d2) exit
  [3] [daikon.chicory.ParameterInfo:d1, daikon.chicory.ParameterInfo:
       d2, daikon.chicory.ReturnInfo:return]
  [1] [daikon.chicory.ParameterInfo:yr]
```

In the '`trace`' file, DynComp outputs comparability sets as trees, structured such that each variable in the tree has interacted with its children. The lack of a parent-child relationship between two variables in a set does not imply anything about whether they interacted. The above '`.decls`' output corresponds to the following in '`trace`':

```
Daikon Traced Tree for Year.compute(I yr, I d1, I d2) enter

daikon.chicory.ParameterInfo:d1

daikon.chicory.ParameterInfo:d2

daikon.chicory.ParameterInfo:yr


Daikon Traced Tree for Year.compute(I yr, I d1, I d2) exit

daikon.chicory.ParameterInfo:d2
--daikon.chicory.ParameterInfo:d1 ([Year:compute()-11])
```

```
--daikon.chicory.ReturnInfo:return ([Year:compute()-11])

daikon.chicory.ParameterInfo yr
```

The file here shows that `d1`, `d2`, and the return value of the `compute` method are in the same comparability set; this is correct, as they are all of the abstract type 'days'. The variable `yr` is in its own comparability set; it has abstract type 'year', and so is not comparable to the other variables. In addition, the structure of the `[d1, d2, return]` set shows that at some point, `d1` interacted with `d2`, and that `d2` interacted with `return`. The absence of a `d1 -- return` edge does not imply that `d1` and `return` never interacted directly.

In addition, non-root nodes in the '`trace`' trees can indicate a list of class names, method names, and line numbers at which values interacted, resulting in comparability between the preceding child node and its parent. In the above example, `d1` interacted with `d2` on line 11 of the `compute` method of the `Year` class.

Duplicate values in this list represent the results of separate calls to another method which each of the relevant variables. For example, if `main` had calls `compute(year, summerDays, winterDays)` and `compute(year,schoolDays,breakDays)`, then for `main` we might see

```
daikon.chicory.FieldInfo:summerDays
--daikon.chicory.FieldInfo:winterDays([Year:compute()-11])
--daikon.chicory.FieldInfo:schoolDays([Year:compute()-11, Year:compute()-11])
----daikon.chicory.FieldInfo:breakDays([Year:compute()-11])
```

Empty lists indicate that no non-assignment interactions occurred in the series of interactions connecting the two variables.

Elements of these lists are essentially parts of stack traces. The maximum number of stack trace levels displayed is set by `--trace-line-depth`, which is equal to 1 by default.

For these files, DynComp also has a `--abridged-vars` option that replaces text like `daikon.chicory.ParameterInfo:d2` with text like `Parameter d2` in the '`cset`' and '`trace`' files. It writes `this` instead of `daikon.chicory.ThisObjInfo:this`; and `return` instead of `daikon.chicory.ReturnInfo:return`. This option is off by default, but can be turned on with `--abridged-vars`.

7.2.1 Instrumenting the JDK with DynComp

If you did not already do so when installing Daikon (in Section 2.2 [Complete installation], page 2), follow the instructions here to build an instrumented copy of the JDK. Use the following command in '`$DAIKONDIR`':

```
make -C java dcomp_rt.jar
```

Make sure the `JDKDIR` environment variable is set to the directory containing your JDK. This command instruments the classes in the '`rt.jar`' file of the JDK, and creates a new file, '`dcomp_rt.jar`', in the '`java`' directory.

On MacOS, there is not normally a JDK directory. Instead, the `ORIG_RT` variable must be set to specify the location of the input JDK jar file (`classes.jar` in MacOS). Normally the file is found in `/System/Library/Frameworks` under the appropriate Java version. The following example is for the standard install of Java 1.5 on MacOS:

```
export ORIG_RT=/System/Library/Frameworks/JavaVM.framework/Versions/1.5.0/Classes/cl
```

BuildJDK requires 20-40 minutes to complete and uses 1024 MB of memory. Regular progress indicators are printed to standard output.

You can ignore warnings issued during the instrumentation process, so long as the make target itself completes normally.

There are a small number of methods in the JDK that DynComp is currently unable to instrument. The names of these methods will be printed at the end of instrumentation. This is not a problem unless your application calls these methods (directly or indirectly). If one of those methods is called, a NoSuchMethodException will be generated when the call is attempted.

7.2.2 DynComp options

DynComp is invoked as follows:

`java daikon.DynComp` *`dyncomp-args classname args`*

where

`java` *`classname args`*

is a valid invocation of Java.

This section lists the optional command-line arguments to DynComp, which appear before the *classname* on the DynComp command line.

'`--verbose`'
: Print information about the classes being processed.

'`--debug`'
: Dump the instrumented classes to '`/tmp/$USER/bin`'.

'`--debug-dir`'
: The directory in which to dump instrumented class files (only if '`--debug`' is specified). Defaults to '`debug`' in the current working directory.

'`--output-dir=`*`dir`*'
: The directory in which to create output files. Defaults to the current working directory.

'`--decl-file=`*`file`*'
: Output filename for decl file suitable for input to Daikon. Defaults to '*`target_program`*`.decls-DynComp`'.

'`--no-cset-file`'
: When this switch is on, output of the '`cset`' file is suppressed.

'`--compare-sets-file=`*`file`*'
: Output filename for a more easily human-readable file summarizing comparability sets. The default behavior is to print to standard output. This switch has no effect if '`--no-cset-file`' is specified on the command line.

'`--trace-sets-file=`*`file`*'
: Output filename for a human-readable file showing some of the interactions that occurred. Default behavior is to not create the file.

'`--trace-line-depth=`*`n`*'
: Controls size of the stack displayed in tracing the interactions that occurred. Default behavior is to only display one element in the stack – that is, display

at most the topmost function on the stack when the interaction occurred. This switch has no effect if '--trace-sets-file' is not specified, or is null.

'--abridged-vars'
: When this switch is on, DynComp abridges the variables printed in the files specified by '--compare-sets-file' and '--trace-sets-file'. For example, DynComp with output "Field foo" instead of "dyncomp.chicory.FieldInfo:MyClass.foo". In particular, it replaces "dyncomp.chicory.ReturnInfo:return" with "return" and "dyncomp.chicory.ThisObjInfo:this" with "this".

'--ppt-select-pattern=*regex*'
: Only emit program points that match *regex*. Specifically, a program point is considered to match *regex* if the fully qualified class name, the method name, or the program point name matches *regex*. The behavior of this switch is the same as in Chicory (see Section 7.1.1.1 [Program points in Chicory output], page 89).

  This option can be specified multiple times, and may be used in conjunction with '--ppt-omit-pattern'. If a program point matches both a select pattern and an omit pattern, it is omitted.

'--ppt-omit-pattern=*regex*'
: Suppress program points that match *regex*. Specifically, a program point is considered to match *regex* if the fully qualified class name, the method name, or the program point name matches *regex*. The behavior of this switch is the same as in Chicory (see Section 7.1.1.1 [Program points in Chicory output], page 89).

  This option can be specified multiple times, any may be used in conjunction with '--ppt-select-pattern'. If a program point matches both a select pattern and an omit pattern, it is omitted.

'--no-primitives'
: Don't track primitives. When this switch is on, DynComp only tracks the comparability of object references; primitive values are ignored. Using this switch can greatly improve DynComp's runtime if you are not interested in primitive values.

'--no-jdk'
: When this switch is on, DynComp runs with an uninstrumented JDK, and the '--rt-file' switch is ignored.

'--rt-file'
: Specifies the location of the instrumented JDK (see Section 7.2.1 [Instrumenting the JDK with DynComp], page 97). This option is rarely necessary, because if '--rt-file' is not specified, DynComp will search for a file named 'dcomp_rt.jar' along the classpath, and in '$DAIKONDIR/java'. Both this file and the current classpath are placed on the boot classpath for DynComp's execution.

'--std-visibility'
: When this switch is on, DynComp traverses exactly those fields that are visible from a given program point. For an example, see Section 7.1.1.2 [Variables in Chicory output], page 90.

'--nesting-depth=*n*'
: Depth to which to examine structure components (default 2). This parameter determines which variables the front end causes to be output at runtime. For an example, see Section 7.1.1.2 [Variables in Chicory output], page 90.

7.2.3 Instrumentation of Object methods

DynComp is unable to directly instrument methods of the class `Object`, such as `clone` and `equals`. DynComp uses a few tricks, described here in brief, to track comparability in these methods.

Calls such as `o1.equals(o2)` are replaced with calls to a static method in DynComp, `dcomp_equals(o1, o2)`. This static method dynamically determines whether or not `o1` is an instance of a class that has been instrumented by DynComp; every such class implements the interface `DCompInstrumented`. If so, it attempts to invoke the instrumented version of the `equals` method for `o1`. If not, or if `o1` has not overridden the `equals` method from `Object`, then no instrumented version exists, so the uninstrumented version is invoked.

In either case, the references `o1` and `o2` are considered to be comparable. In a future release, we will provide a command-line switch to customize this behavior.

The `clone` method operates in a similar manner, choosing dynamically to invoke the instrumented method or the uninstrumented method. In the case of `clone`, the methods are invoked via reflection. In either case, the object being cloned and the resulting clone are made comparable to each other. Again, we will provide a switch to customize this behavior in a future release.

7.2.4 Known bugs and limitations

- Reflection finds the original, uninstrumented code, and so DynComp may not accurately instrument code that uses reflection.
- Instrumentation of the `clone()` method may fail on particular invocations within private classes in the JDK.

7.3 C/C++ front end Kvasir

Daikon's front end for C and C++, named Kvasir, executes C and C++ programs and creates data trace ('`.dtrace`') files of variables and their values by examining the operation of the binary at runtime. Kvasir is named after the Norse god of knowledge and beet juice. It is built upon the Fjalar dynamic analysis framework for C and C++ programs (available at `http://pag.csail.mit.edu/fjalar/`, but already included in the Daikon distribution).

To use Kvasir, first compile your program using the DWARF-2 debugging format (e.g., supply the '`-gdwarf-2`' option to `gcc`) and without optimizations (e.g., supply the '`-O0`' option to `gcc`). Then, prefix your command line by `kvasir-dtrace`. For example, if you normally run your program with the command

```
./program -option input.file
```

then instead use the command

```
kvasir-dtrace ./program -option input.file
```

to run your program and create a data trace file '`daikon-output/program.dtrace`', which can be fed as input into Daikon. You can perform this step multiple times to create multiple data trace files for Daikon. You can also run Daikon without creating an intermediate data trace file; see Section 7.3.7 [Online execution], page 119.

For information about installing Kvasir, see Section 7.3.8 [Installing Kvasir], page 121. Kvasir only works under Linux running on an x86 or x86-64 processor; for full details, see Section 7.3.9 [Kvasir limitations], page 121. For information about how to create an instrumenter for C that works on non-Linux or non-x86 platforms, see section "Instrumenting C programs" in *Daikon Developer Manual*.

7.3.1 Using Kvasir

Before using Kvasir, you must compile your program compile and link your program normally, with two exceptions:

- Do not use optimization. Remove any optimization flags, such as '`-O`' or '`-O2`', and any flags that affect calling conventions, such as '`-fomit-frame-pointer`'.
- Include debugging information, by supplying the '`-g`' flag. The debugging information must be in the DWARF-2 format. DWARF-2 is the default format for debugging information in GCC 3 and later, and otherwise is produced by supplying the '`-gdwarf-2`' command line option.

In the second step of using Kvasir, run your program as you normally would, but prepend the command `kvasir-dtrace` to the beginning. For instance, if you normally run your program with the command

```
./myprogram -option input.file
```

just say

```
kvasir-dtrace ./myprogram -option input.file
```

As well as running your program (more slowly than usual), this command also creates a directory '`daikon-output`' in the current directory containing a '`program.dtrace`' file suitable as input to Daikon.

Kvasir's first argument, the program name, should be given as a pathname, as shown above. If you usually just give a program name that is not in the current directory but is found in your path, you may need to modify your command to specify a pathname. For example:

```
kvasir-dtrace `which myprogram` -option input.file
```

You may supply options to Kvasir before the argument that is the name of your program (see Section 7.3.2 [Kvasir options], page 101).

7.3.2 Kvasir options

To see a complete list of options, run this command: `kvasir-dtrace --help`

Output file format:

'`--decls-file=`*filename*'
: Write the '`.decls`' file listing the names of functions and variables (called declarations) to the specified file name. This forces Kvasir to generate separate '`.decls`' and '`.dtrace`' files instead of outputting everything to the '`.dtrace`' file, which is the default behavior. If only a '`.dtrace`' file is created (default behavior), then it contains both variable declarations and a trace of values. If separate '`.decls`' and '`.dtrace`' files are created, then the '`.decls`' file contains declarations and the '`.dtrace`' file contains the trace of values.

'`--decls-only`'
: Exit after writing the '`.decls`' file; don't run the program or generate trace information. Since the '`.decls`' file is the same for any run of a program, it can be generated once and then reused on later runs, as long as no new program points are added and each program point has the same set of variables.

'`--dtrace-file=`*filename*'
: Write the '`.dtrace`' trace file to the specified file name. The default is '`daikon-output/`*programname*`.dtrace`', where *programname* is the name of the program. A filename of '`-`' may be used to specify the standard output; in this case, the regular standard output of the program will be redirected back to the terminal ('`/dev/tty`'), to avoid intermixing it with the trace output. If the given filename ends in '`.gz`', then '`--dtrace-gzip`' is enabled and the '`.dtrace`' file will be compressed.

'`--dtrace-no-decs`'
: By default, the '`.dtrace`' file contains both a list of variable declarations followed by a trace of variable values (see section "File formats" in *Daikon Developer Manual*). If this option is used, then variable declarations are not outputted in the .dtrace file. This option is equivalent to '`--decls-file=/dev/null`', except that it runs faster. This is useful when you want to generate one copy of the declarations in the .decls file using '`--decls-only`', generate many '`.dtrace`' files from different program runs, and then feed 1 '`.decls`' and several '`.dtrace`' files into Daikon.

'`--dtrace-append`'
: Append new trace information to the end of an existing '`.dtrace`' file. The default is to overwrite a pre-existing '`.dtrace`' file. When this option is used, no declaration information is written because it is assumed that the existing .dtrace file already contains all declarations (Daikon does not accept duplicate declarations).

'`--dtrace-gzip`'
: Compress trace information with the gzip program before writing it to the '`.dtrace`' file. You must have the gzip program available.

'`--output-fifo`'
: Create the output '`.dtrace`' file as a FIFO (also known as a "named pipes"). Kvasir will then open first the '`.decls`' FIFO and then the '`.dtrace`' FIFO, blocking until another program (such as Daikon) reads from them. Using FIFOs for the output of Kvasir avoids the need for large trace files, but FIFOs are not supported by some file systems, including AFS.

'`--program-stdout=`*`filename`*'
'`--program-stderr=`*`filename`*'
: Redirect the standard output (respectively, standard error) stream of the program being traced to the specified path. By default, the standard output and standard error streams will be left pointing to the same locations specified by the shell, except that if '`--dtrace-file=-`' is specified, then the default behavior is as if '`--program-stdout=/dev/tty`' were specified, since mixing the program's output and Kvasir's trace output is not advisable. If the same filename is given for both options, the streams will be interleaved in the same way as if by the Bourne shell construction `2>&1`.

  Also, as in the shell, *filename* can be an ampersand followed by an integer, to redirect to a numbered file descriptor. For instance, to redirect the program's standard output and error, and Kvasir's standard error, to a single file, you can say '`--program-stdout='&2' --program-stderr='&2' 2>`*`filename`*'.

Selective program point and variable tracing:

'`--ppt-list-file=`*`filename`*'
'`--var-list-file=`*`filename`*'
: Trace only the program points (respectively, variables) listed in the given file. Other program points (respectively variables) will be omitted from the '`.decls`' and '`.dtrace`' files. A convenient way to produce such files is by editing the output produced by the '`--dump-ppt-file`' (respectively, '`--dump-var-file`') option described below. (see Section 7.3.4 [Tracing only part of a program], page 109 section for detailed instructions on using these options.)

'`--dump-ppt-file=`*`filename`*'
'`--dump-var-file=`*`filename`*'
: Print a list of all the program points (respectively all the variables) in the program to the specified file. An edited version of this file can then be used with the '`--ppt-list-file`' (respectively '`--var-list-file`') option. (see Section 7.3.4 [Tracing only part of a program], page 109 section for detailed instructions on using these options.) Note: Do not use these options with the '`--with-dyncomp`' option because the behavior is undefined. Running Kvasir with these options will initialize but not actually execute the target program, so the dynamic comparability analysis cannot be performed in the first place.

'`--ignore-globals`'
: Omit any global or static variables from the '`.decls`' and '`.dtrace`' files. Leaving these out can significantly improve Kvasir and Daikon's performance, at the expense of missing properties involving them. The default is to generate trace information for global and static variables.

'`--ignore-static-vars`'
: Omit any static variables but generate trace information for global variables in the '`.decls`' and '`.dtrace`' files.

'`--all-static-vars`'
: Output all static variables at all program points in the '`.decls`' and '`.dtrace`' files. By default, file-static variables are only outputted at program points for

functions that are defined in the same file (compilation unit) as the variable, and static variables declared within a particular function are only outputted at program points for that function. These heuristics decrease clutter in the output without greatly reducing precision because functions have no easy way of modifying variables that are not in-scope, so it is often not useful to output those variables. This option turns off these heuristics and always outputs static variables at all program points.

Other options affecting the amount of output Kvasir produces:

'`--flatten-arrays`'
: This option forces the flattening of statically-sized arrays into separate variables, one for each element. For example, an array *foo* of size 3 would be flattened into 3 variables: *foo[0]*, *foo[1]*, *foo[2]*. By default, Kvasir flattens statically-sized arrays only after it has already exhausted the one level of sequences that Daikon allows in the .dtrace output format (e.g. an array of structs where each struct contains a statically-sized array).

'`--array-length-limit=`*N*'
: Only visit at most the first *N* elements of all arrays. This can improve performance at the expense of losing coverage; it is often useful for tracing selected parts of programs that use extremely large arrays or memory buffers.

'`--output-struct-vars`'
: This option forces Kvasir to output .decls and .dtrace entries for struct variables. By default, Kvasir ignores struct variables because there is really no value that can be meaningfully associated with these variables. However, some tools require struct variables to be outputted, so we have included this option. Struct variables are denoted by a `# isStruct=true` annotation in their declarations.

'`--nesting-depth=`*N*'
: For recursively-defined structures (structs or classes with members that are structs or classes or pointers to structs or classes of *any* type), *N* (an integer between 0 and 100) specifies approximately how many levels of pointers to dereference. This is useful for controlling the output of complex data structures with many references to other structures. The default is 2.

'`--struct-depth=`*N*'
: For recursively-defined structures (structs or classes with members that are pointers to the *same* type of struct or class), *N* (an integer between 0 and 100) specifies approximately how many levels of pointers to dereference. This is useful for controlling the output of linked lists and trees. The default is 4. If you are trying to traverse deep into data structures, try adjusting the '`--struct-depth`' and '`--total-depth`' options until Kvasir traverses deep enough to reach the desired variables.

Section 7.3.5 [Pointer type disambiguation], page 112:

'`--disambig-file=`*`filename`*'
: Specifies the name of the pointer type disambiguation file (see Section 7.3.5 [Pointer type disambiguation], page 112). If this file exists, Kvasir uses it to

make decisions about how to output the referents of pointer variables. If the file does not exist, then Kvasir creates it. This file may then be edited and used on subsequent runs. This option initializes but does not fully execute the target program (unless it is run with the '`--smart-disambig`' option).

'`--disambig`'
: Tells Kvasir to create or read pointer type disambiguation (see Section 7.3.5 [Pointer type disambiguation], page 112) with the default filename, which is '*myprog*`.disambig`' in the same directory as the target program, where *myprog* is the name of the target program. This is equivalent to '`--disambig-file='`*myprog*`.disambig'`'.

'`--smart-disambig`'
: This option should be used in addition to either the '`--disambig`' or '`--disambig-file`' options (it does nothing by itself). If the .disambig file specified by the option does not exist, then Kvasir executes the target program, observes whether each pointer refers to either one element or an array of elements, and creates a disambiguation file that contains suggestions for the disambiguation types of each pointer variable. This potentially provides more accuracy than using either the '`--disambig`' or '`--disambig-file`' options alone, but at the expense of a longer run time. (If the .disambig file already exists, then this option provides no extra functionality.)

'`--func-disambig-ptrs`'
: By default, Kvasir treats all pointers as arrays when outputting their contents. This option forces Kvasir to treat function parameters and return values that are pointers as pointing to single values. However, all pointers nested inside of data structures pointed-to by parameters and return values are still treated as arrays. This is useful for outputting richer data information for functions that pass parameters or return values via pointers, which happens often in practice.

'`--disambig-ptrs`'
: By default, Kvasir treats all pointers as arrays when outputting their contents. This option forces Kvasir to treat all pointers as pointing to single values. This is useful when tracing nested structures with lots of pointer fields which all refer to one element.

Section 7.3.3 [DynComp for C/C++], page 107:

'`--with-dyncomp`'
: Run Kvasir with the DynComp dynamic comparability analysis tool to determine which variables have the same abstract type. Variable comparability information can improve the performance of Daikon and allow it to generate a more focused and relevant set of invariants. Because it is not available until the end of execution, comparability information is always written to a separate '`.decls`' file (in the format specified in the section "Program point declarations" in *Daikon Developer Manual* section), as if the '`--decls-file`' option had been specified ('`--decls-file`' can still be used to control the name of the file). This file must be provided to Daikon along with the '`.dtrace`' file. This option may also be used with '`--decls-only`' to only generate a '`.decls`' file without a '`.dtrace`'.

'`--no-dyncomp-gc`'
: By default, DynComp runs with a garbage collector for the tag metadata that it uses, but this can cause your program to slow down if it runs too often. This option turns off the garbage collector. This is not recommended for long program runs, because without the garbage collector, it will likely run out of memory.

'`--gc-num-tags=`*N*'
: The DynComp garbage collector runs once after every 10,000,000 tags have been assigned. This option tells the garbage collector to run once after every *N* tags have been assigned. Making the value of *N* larger allows your program to run faster (because the garbage collector runs less frequently), but may cause your program to run out of memory as well. Making the value of *N* too small may cause your program to never terminate if *N* is smaller than the total number of tags that your program uses in steady state. You will probably need to experiment with tweaking this value in order to get DynComp to work properly.

'`--dyncomp-fast-mode`'
: This option applies an approximation for handling literal values which greatly speeds up the performance of DynComp and drastically lowers its memory usage, but at the expense of a slight loss in precision of the generated comparability sets. If you cannot get DynComp to successfully run on a large program, even after tweaking '`--gc-num-vars`', try turning on this option.

'`--dyncomp-detailed-mode`'
: This option runs a more detailed (but more time- and space-intensive) algorithm for tracking variable comparability. It takes O(n^2) time and space, whereas the default algorithm takes roughly O(n) time and space. However, it can produce more precise results. Despite its name, this mode can be used together with '`--dyncomp-fast-mode`' to run the more precise algorithm but still use an approximation for handling literal values. (This mode is still experimental and not well-tested yet.)

'`--separate-entry-exit-comp`'
: The default behavior for DynComp is to generate the same comparability numbers for Daikon variables at each pair of function entrance and exit program points. If this option is used, then DynComp keeps track of comparability separately for function entrances and exits, which can lead to more accurate results, but sometimes generates output .decls files that Daikon cannot accept.

'`--dyncomp-dataflow-only`'
: When DynComp is operating in this mode, no binary operations qualify as interactions between values. Thus, DynComp only tracks dataflow.

'`--dyncomp-dataflow-comp`'
: When DynComp is operating in this mode, the only binary operations that qualify as interactions are comparisons between values (e.g., `x <= y` or `x != y`).

'`--dyncomp-units`'
: When DynComp is operating in this mode, the only binary operations that qualify as interactions are comparisons, addition, subtraction. This ensures that the variables that DynComp groups together into one set all have the same units (e.g., physics units).

Debugging:

'`--xml-output-file=`*`filename`*'
: Outputs a representation of data structures, functions, and variables in the target program to an XML file in order to aid in debugging. These are all the entities that Kvasir tracks for a particular run of a target program, so if you do not see an entity in this XML file, then you should either adjust command-line options or contact us with a bug report.

'`--with-gdb`'
: This pauses the program's execution in an infinite loop during initialization. You can attach a debugger such as `gdb` to the running process by running gdb on '`inst/lib/valgrind/x86-linux/fjalar`' under the Kvasir directory and using the `attach` command.

'`--debug`'
'`--dyncomp-debug`'
: Enable progress messages meant for debugging problems with Kvasir or DynComp. By default, they are disabled. This option is intended mainly for Kvasir's developers.

7.3.3 DynComp dynamic comparability (abstract type) analysis for C/C++

Kvasir comes with the DynComp dynamic comparability analysis tool, which performs dynamic type inference to group variables at each program point into comparability sets (see section "Program point declarations" in *Daikon Developer Manual* for the numeric representation format of these sets). All variables in each comparability set belong to the same "abstract type" of data that the programmer likely intended to represent, which is a richer set of types than the few basic declared types (e.g., int, float) provided by the language. Consider the example below:

```
int main() {
  int year = 2005;
  int winterDays = 58;
  int summerDays = 307;
  compute(year, winterDays, summerDays);
}

int compute(int yr, int d1, int d2) {
  if (yr % 4)
    return d1 + d2;
  else
    return d1 + d2 + 1;
}
```

The three variables in `main()` all have the same C representation type, `int`, but two of them hold related quantities (numbers of days), as can be determined by the fact that they interact when the program adds them, whereas the other contains a conceptually distinct quantity (a year). The abstract types 'day' and 'year' are both represented as `int`, but DynComp can differentiate them with its dynamic analysis. For example, DynComp can infer that `winterDays` and `summerDays` are comparable (belong to the same abstract type) because the program adds their values together within the `compute()` function.

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

Use the '`--with-dyncomp`' option to run Kvasir with DynComp to generate a '`.decls`' file with comparability information along with the usual value trace in the '`.dtrace`' file. Using '`--decls-only`' will only generate the '`.decls`' file without the extra slowdown of writing the '`.dtrace`' file to disk (however, because DynComp must execute the entire program to perform its analysis, the only time saved is I/O time). Other DynComp options are listed in the Section 7.3.2 [Kvasir options], page 101 section. Running Kvasir with DynComp takes more memory and longer time than running Kvasir alone, but remember that DynComp only needs to be run once to generate a '`.decls`' file with comparability information. That one file can be passed into Daikon along with many different '`.dtrace`' files generated during subsequent Kvasir runs without DynComp.

Here is part of the '`.decls`' file generated by running Kvasir with DynComp on the above example:

```
DECLARE
..compute():::ENTER
yr
int # isParam=true
int
1
d1
int # isParam=true
int
2
d2
int # isParam=true
int
2

DECLARE
..compute():::EXIT0
yr
int # isParam=true
int
```

```
1
d1
int # isParam=true
int
2
d2
int # isParam=true
int
2
return
int
int
2
```

The abstract type of 'year' (and its corresponding comparability set) is represented by the number 1 while the abstract type of 'day' is represented by the number 2. DynComp places two variables in the same comparability set when their values interact via program operations such as arithmetic or assignment. Because the parameters `d1` and `d2` were added together, DynComp inferred that those variables were somehow related and put them in the same comparability set. The return value is also related to `d1` and `d2` because it is the result of the addition operation. Notice that `yr` never interacts with any other variables, so DynComp places it into its own comparability set. With this comparability information, Daikon will never attempt to find invariants between `yr` and `d1`/`d2`, which both saves time and memory and eliminates meaningless invariants (the savings are miniscule in this trivial example, but they can be rather dramatic in larger examples).

7.3.4 Tracing only part of a program

When Kvasir is run on a target program of significant size, often times too much output is generated, which causes an enormous performance slowdown of both Kvasir outputting the trace file and also Daikon trying to process the trace file. It is often desirable to only trace a specific portion of the target program, program points and variables that are of interest for a particular invariant detection application. For instance, one may only be interested in tracking changes in a particular global data structure during calls to a specific set of functions (program points), and thus have no need for information about any other program points or variables in the trace file. The '`--ppt-list-file`' and '`--var-list-file`' options can be used to achieve such selective tracing.

The program point list file (abbreviated as '`ppt-list-file`') consists of a newline-separated list of names of functions that the user wants Kvasir to trace. Every name corresponds to both the entrance (`:::ENTER`) and exit (`:::EXIT`) program points for that function and is printed out in the exact same format that Kvasir uses for that function in the trace file (see Section 7.3.1 [Using Kvasir], page 101 section for the program point naming scheme). Here is an example of a '`ppt-list-file`':

```
FunctionNamesTest.cpp.staticFoo(int, int)
..firstFileFunction(int)
..main()
second_file.cpp.staticFoo(int, int)
..secondFileFunction()
```

It is very important to follow this format in the 'ppt-list-file' because Kvasir performs string comparisons to determine which program points to trace. Thus, it is often easier to have Kvasir generate a 'ppt-list-file' file that contains a list of all program points in a target program by using the '--dump-ppt-file' option, and then either comment out (by using the '#' comment character at the beginning of the line) or delete lines in that file for program points not to be traced or create a new 'ppt-list-file' using the names in the Kvasir-generated file. This prevents typos and the tedium of manually typing up program point names. In fact, the 'ppt-list-file' presented in the above example was generated from a C++ test program named FunctionNamesTest by using the following command:

```
kvasir-dtrace --dump-ppt-file=FunctionNamesTest.ppts  ./FunctionNamesTest
```

That file represents all the program points that Kvasir would normally trace. If the user wanted to only trace the main() function, he could comment out all other lines by placing a single '#' character at the beginning of each line to be commented out, as demonstrated here:

```
#FunctionNamesTest.cpp.staticFoo(int, int)
#..firstFileFunction(int)
..main()
#second_file.cpp.staticFoo(int, int)
#..secondFileFunction()
```

When running Kvasir with the '--ppt-list-file' option using this as the 'ppt-list-file', Kvasir only stops the execution of the target program at the entrance and exit of main() in order to output values to the .dtrace file. In order to reduce the file size, when running Kvasir with the '--ppt-list-file' option, the .decls file only contains program point declarations for those listed in the 'ppt-list-file' (..main():::ENTER and ..main():::EXIT in this case) because no other declarations are necessary.

The variable list file (abbreviated as 'var-list-file') contains all of the variables that the user wants Kvasir to output. There is one section for global variables and a section for variables associated with each function (formal parameters and return values). Again, the best way to create a 'var-list-file' is to have Kvasir generate a file with all variables using the '--dump-var-file' option and then modifying that file for one's particular needs by either deleting or commenting out lines (again using the '#' comment character). For example, executing

```
kvasir-dtrace --dump-var-file=FunctionNamesTest.vars  ./FunctionNamesTest
```

will generate the following 'var-list-file' named FunctionNamesTest.vars:

```
----SECTION----
globals
/globalIntArray
/globalIntArray[]
/anotherGlobalIntArray
/anotherGlobalIntArray[]


----SECTION----
FunctionNamesTest.cpp.staticFoo()
x
```

```
y


----SECTION----
..firstFileFunction(int)
blah


----SECTION----
..main()
argc
argv
argv[]
return


----SECTION----
second_file.cpp.staticFoo()
x
y


----SECTION----
..secondFileFunction()
```

The file format is straightforward. Each section is marked by a special string "`----SECTION----`" on a line by itself followed immediately by a line that either denotes the program point name (formatted like how it appears in the .decls and .dtrace files) or the special string "`globals`". This is followed by a newline-delimited list of all variables to be outputted for that particular program point. (Global variables listed in the `globals` section are outputted for all program points.) For clarity, one or more blank lines should separate neighboring sections, although the "`----SECTION----`" string literal on a line by itself is the only required delimiter. If an entire section is missing, then no variables for that program point (or no global variables, if it is the special globals section) are traced.

The variables listed in this file are written exactly as they appear in the .decls and .dtrace file (see Section 7.3.1 [Using Kvasir], page 101 section for the variable naming scheme). In the program that generated the output for the above example, `int* globalIntArray` is a global integer pointer variable. For that variable, Kvasir generates two Daikon variables: `/globalIntArray` to represent the hashcode pointer value, and `/globalIntArray[]` to represent the array of integers referred-to by that pointer. The latter is a derived-variable that can be thought of as the child of `/globalIntArray`. If the entry for `/globalIntArray` is commented-out or missing, then Kvasir will not output any values for `/globalIntArray` or for any of its children, which in this case is `/globalIntArray[]`. If a struct or struct pointer variable is commented-out or missing, then none of its members are traced. Thus, a general rule about variable entries in the '`var-list-file`' is that if a parent variable is not present, then neither it nor its children are traced.

```
record
record->entries[1]
record->entries[1]->list
record->entries[1]->list->head
record->entries[1]->list->head->magic
```

For example, if you wanted to trace the value of the `magic` field nested deep within several layers of structs and arrays, it would not be enough to merely list this variable in the '`var-list-file`'. You would need to list all variables that are the parents of this one, as indicated by their names. This can be easily accomplished by creating a file with '`--dump-var-file`' and cutting out variable entries, taking care to not cut out entries that are the parents of entries that you want to trace.

In order to limit both the number of program points traced as well as the variables traced at those program points, the user can run Kvasir with both the '`--ppt-list-file`' and '`--var-list-file`' options with the appropriate '`ppt-list-file`' and '`var-list-file`', respectively. The '`var-list-file`' only needs to contain a section for global variables and sections for all program points to be traced because variable listings for program points not to be traced are irrelevant (their presence in the '`var-list-file`' does not affect correctness but does cause an unnecessary performance and memory inefficiency).

If the '`--dump-var-file`' option is used in conjunction with the '`--ppt-list-file`' option, then the only sections generated in the '`var-list-file`' will be the global section and sections for all program points explicitly mentioned in the '`ppt-list-file`'. This is helpful for generating a smaller '`var-list-file`' for use with an already-existent '`ppt-list-file`'.

7.3.5 Pointer type disambiguation

Kvasir permits users (or external analyses) to specify whether pointers refer to arrays or to single values, and optionally, to specify the type of a pointer (see Section 7.3.5.1 [Pointer type coercion], page 113). For example, in

```
void sum(int* array, int* result) { ... }  // definition of "sum"
...
int a[40];
int total;
...
sum(a, &total);        // use of "sum"
```

the first pointer parameter refers to an array while the second refers to a single value. Kvasir (and Daikon) should treat these values differently. For instance, `*array` is better printed as `array[]`, an array of integers, and `result[]` isn't a sensible array at all, even though in C `result[0]` is semantically identical to `*result`. By default, Kvasir treats all pointers as referencing arrays. For instance, it would print `result[]` rather than `result[0]` and would indicate that the length of array `result[]` is always 1. In order to improve the formatting of Daikon's output (and to speed it up), you can indicate to Kvasir that certain pointers refer to single elements rather than to arrays. For an example, see Section 7.3.5.2 [Pointer type disambiguation example], page 114).

Information about whether each pointer refers to an array or a single element can be specified in a "disambig file" that resides in the same directory as the target program (by default). The '`--disambig`' option instructs Kvasir to read this file if it exists. (If it does not exist, Kvasir produces the file automatically and, if invoked along with the '`--smart-disambig`' option, heuristically infers whether each pointer variable refers to single or multiple elements. Thus, users can edit this file for use on subsequent runs rather than having to create it from scratch.) The disambig file lists all the program points and user-defined types, and under each, lists certain types of variables along with their custom disambiguation types as shown below. The list of disambiguation options is:

1. For variables of type `char` and `unsigned char`:
   1. 'I': an integer, signed for `char` and unsigned for `unsigned char`. (Default)
   2. 'C': a single character, output as a string.
2. For pointers to (or arrays of) `char` and `unsigned char`:
   1. 'S': a string, possibly zero-terminated. (Default)
   2. 'C': a single character, output as a string.
   3. 'A': an array of integers.
   4. 'P': a single integer.
3. For pointers to (or arrays of) all other variable types (if invoked along with '`--smart-disambig`', Kvasir automatically infers a default 'A' or 'P' for each variable during the generation of a '`.disambig`' file):
   1. 'A': an array. (Default) (For an array of structs, an array will be output for each scalar field of the struct. Aggregate children (arrays, other structs) will not be output.)
   2. 'P': a pointer to a single element. (For a pointer to a struct, each field will be output as a single instance, and child aggregate types will be output recursively. This extra information obtained from struct pointers is a powerful consequence of pointer type disambiguation. This will be the default if the '`--disambig-ptrs`' option is used.)

The disambig file that Kvasir creates contains a section for each function, which can be used to disambiguate parameter variables visible at that function's entrance program point and parameter and return value variables visible at that function's exit program point. It also contains a section for every user-defined struct/class, which can be used to disambiguate member variables of that struct/class. Disambiguation information entered here will apply to all instances of a struct/class of that type, at all program points. There is also a section called "globals", which disambiguates global variables which are output at every program point. The entries in the disambig file may appear in any order, and whole entries or individual variables within a section may be omitted. In this case, Kvasir will retain their default values.

7.3.5.1 Pointer type coercion

In addition to specifying whether a particular pointer refers to one element or to an array of elements, the user can also specify what type of data a pointer refers to. This type coercion acts like an explicit type cast in C, except that it only works on struct/class types and not on primitive types. This feature is useful for traversing inside of data structures with generic `void*` pointer fields. Another use is to cast a pointer from one that refers to a 'super class' to one that refers to a 'sub class'. This structural equivalence pattern is often found in C programs that emulate object orientation. To coerce a pointer to a particular type, simply write the name of the struct type after the disambiguation letter (e.g., A, P, S, C, I) in the '`.disambig`' file:

```
----SECTION----
function: ..view_foo_and_bar()
f
P foo
b
P bar
```

Without the type coercion, Kvasir cannot print out anything except for a hashcode for the two `void*` parameters of this function:

```
void view_foo_and_bar(void* f, void* b);
```

With type coercion, though, Kvasir treats `f` as a `foo*` and `b` as `bar*` and can traverse inside of them. Of course, if those are not the true runtime types of the variables, then Kvasir's output will be meaningless.

Due to the use of typedefs, there may be more than one name for a particular struct type. The exact name that you need to write in the '`.disambig`' file is the one that appears in that file after the `usertype` prefix. Note that if a struct does not have any pointer fields, then there will be no `usertype` section for it in the '`.disambig`' file. In that case, try different names for the struct if necessary until Kvasir accepts the name (names are all one word long; you will never have to write `struct foo`). There should only be at most a few choices to make. If the coercion if successful, Kvasir prints out a message in the following form while it is processing the '`.disambig`' file:

```
  .disambig: Coerced variable f into type 'foo'
  .disambig: Coerced variable b into type 'bar'
```

One more caveat about type coercion is that you can currently only coerce pointers into types that at least one variable in the program (e.g., globals, function parameters, struct fields) belongs to. It is not enough to merely declare a struct type in your source code; you must have a variable of that type somewhere in your program. This is a limitation of the current implementation, but it should not matter most of the time because programs rarely have struct declarations with no variables that belong to that type. If you encounter this problem, you can simply create a global variable of a certain type to make type coercion work.

7.3.5.2 Pointer type disambiguation example

This example demonstrates the power of pointer type disambiguation in creating more accurate Daikon output. Consider this file:

```
struct record {
  char* name;     // Initialize to: "Daikon User"
  int numbers[5]; // Initialize to: {5, 4, 3, 2, 1}
};

void foo(struct record* bar) {
  int i;
  for (i = 0; i < 5; i++) {
```

```
    bar->numbers[i] = (5 - i);
  }
}

int main() {
  char* myName = "Daikon User";
  struct record baz;
  baz.name = myName;
  foo(&baz);
}
```

In `foo()`, `bar` is a pointer to a `record` struct. By inspection, it is evident that in this program, `bar` only refers to one element: `&baz` within `main`. However, by default, Kvasir assumes that `bar` is an array of `record` structs since a C pointer contains no information about how many elements it refers to. Because Kvasir must output `bar` as an array and `bar->numbers` is an array of integers, it "flattens" `bar->numbers` into 5 separate arrays named `bar->numbers[0]` through `bar->numbers[4]` and creates fairly verbose output. This is a direct consequence of the fact that Daikon can only handle one layer of sequences (it cannot handle arrays within arrays, i.e., multi-dimensional arrays).

Here is part of the Daikon output for this program:

```
===========================================================================
..foo():::ENTER
bar has only one value
bar[].name == [Daikon User]
bar[].name elements == "Daikon User"
===========================================================================
..foo():::EXIT
size(bar[]).numbers[0] == size(bar[]).numbers[0][0]
size(bar[]).numbers[0] == size(bar[]).numbers[1]
size(bar[]).numbers[0] == size(bar[]).numbers[1][0]
size(bar[]).numbers[0] == size(bar[]).numbers[2]
size(bar[]).numbers[0] == size(bar[]).numbers[2][0]
size(bar[]).numbers[0] == size(bar[]).numbers[3]
size(bar[]).numbers[0] == size(bar[]).numbers[3][0]
size(bar[]).numbers[0] == size(bar[]).numbers[4]
size(bar[]).numbers[0] == size(bar[]).numbers[4][0]
bar[].name == [Daikon User]
bar[].name elements == "Daikon User"
bar[].numbers[0] contains no nulls and has only one value, of length 1
bar[].numbers[0] elements has only one value
bar[].numbers[0][0] == [5]
bar[].numbers[0][0] elements == 5
bar[].numbers[1] contains no nulls and has only one value, of length 1
bar[].numbers[1] elements has only one value
bar[].numbers[1][0] == [4]
bar[].numbers[1][0] elements == 4
bar[].numbers[2] contains no nulls and has only one value, of length 1
```

```
bar[].numbers[2] elements has only one value
bar[].numbers[2][0] == [3]
bar[].numbers[2][0] elements == 3
bar[].numbers[3] contains no nulls and has only one value, of length 1
bar[].numbers[3] elements has only one value
bar[].numbers[3][0] == [2]
bar[].numbers[3][0] elements == 2
bar[].numbers[4] contains no nulls and has only one value, of length 1
bar[].numbers[4] elements has only one value
bar[].numbers[4][0] == [1]
bar[].numbers[4][0] elements == 1
size(bar[]).numbers[0] == 1
bar[].numbers[4][0] elements == size(bar[]).numbers[0]
size(bar[]).numbers[0] in bar[].numbers[4][0]
```

This is a bit verbose due to the fact that Kvasir treats `bar` like an array by default when it actually only points to one element. However, by running Kvasir with the '`--disambig`' option, we create the '*myprog*`.disambig`' file, which we can then edit and feed back to Kvasir to change how the pointer is treated. (We run Kvasir twice on the same program, but we edit the '`.disambig`' file in between the runs.)

```
kvasir-dtrace ...options... --disambig --smart-disambig myprog
```

This creates the '*myprog*`.disambig`' file. It contains, at the top:

```
----SECTION----
function: ..foo()
bar
P
```

This means that at the program points corresponding to the entry and exit of `foo()`, the variable `bar` is treated as a '`P`'ointer type. Since we have used the '`--smart-disambig`' option, Kvasir automatically inferred Pointer instead of Array for `bar` because it observed that `bar` only pointed to one element during the execution of the target program which generated the '`.disambig`' file. This heuristic allows users to use '`.disambig`' files more effectively with less manual editing. Without '`--smart-disambig`', Kvasir does not execute the program to make such inferences, which allows .disambig files to be generated faster, but leaves the default disambiguation types for all entries (in this case, `bar` would have the default array type of `'A'`).

Then, running Kvasir again with the '`--disambig`' option causes Kvasir to open the existing '*myprog*`.disambig`' file, read the definitions, and alter the output accordingly:

```
kvasir-dtrace ...options... --disambig myprog
```

This tells Kvasir to output `bar` as a '`P`'ointer to a single element, which in turn causes Daikon to generate a much more concise set of invariants. Notice that `bar->numbers` no longer has to be "flattened" because `bar` is now a pointer to one struct, so Daikon can recognize `bar->numbers` as a single-dimensional array (Daikon uses a Java-like syntax, replacing the arrow '`->`' symbol with a dot, so it actually outputs `bar.numbers`).

```
===========================================================================
..foo():::ENTER
bar has only one value
```

```
bar.name == "Daikon User"
===========================================================================
..foo():::EXIT
bar.name == "Daikon User"
bar.numbers has only one value
bar.numbers[] == [5, 4, 3, 2, 1]
size(bar.numbers[]) == 5
bar.name == orig(bar.name)
size(bar.numbers[]) in bar.numbers[]
size(bar.numbers[])-1 in bar.numbers[]
```

7.3.5.3 Using pointer type disambiguation with partial program tracing

It is possible to use pointer type disambiguation while only tracing selected program points and/or variables in a target program, combining the functionality described in the Section 7.3.5 [Pointer type disambiguation], page 112 and Section 7.3.4 [Tracing only part of a program], page 109 sections. This section describes the interaction of the '`ppt-list-file`', '`var-list-file`', and .disambig files.

The interaction between selective program point tracing (via the '`ppt-list-file`') and pointer type disambiguation is fairly straightforward: If the user creates a .disambig file while running Kvasir with a '`ppt-list-file`' that only specifies certain program points, the generated .disambig file will only contain sections for those program points (as well as the global section and sections for each struct type). If the user reads in a .disambig file while running Kvasir with a '`ppt-list-file`', then disambiguation information is applied for all variables at the program points to be traced. This can be much faster and generate a much smaller disambiguation file, one that only contains information about the program points of interest.

The interaction between selective variable tracing (via the '`var-list-file`') and pointer type disambiguation is a bit more complicated. This is because the '`var-list-file`' lists variables as they appear in the .decls and .dtrace files, but using a .disambig file can actually change the way that variable names are printed out in the .decls and .dtrace files. For example, consider the test program from the Section 7.3.5.2 [Pointer type disambiguation example], page 114. The `struct record* bar` parameter of `foo()` is treated like an array by default. Hence, the .decls, .dtrace, and '`var-list-file`' will list the following variables derived from this parameter:

```
----SECTION----
..foo()
bar
bar[].name
bar[].numbers[0]
bar[].numbers[0][0]
bar[].numbers[1]
bar[].numbers[1][0]
bar[].numbers[2]
bar[].numbers[2][0]
bar[].numbers[3]
```

```
bar[].numbers[3][0]
bar[].numbers[4]
bar[].numbers[4][0]
```

However, if we use a disambiguation file to denote `bar` as a pointer to a single element, then the .decls and .dtrace files will instead list the following variables:

```
----SECTION----
..foo()
bar
bar->name
bar->numbers
bar->numbers[]
```

Notice how the latter variable list is more compact and reflects the fact that `bar` is a pointer to a single struct. Thus, the flattening of the `numbers[5]` static array member variable is no longer necessary (it was necessary without disambiguation because Daikon does not support nested arrays of arrays, which can occur if `bar` were itself an array since `numbers[5]` is already an array).

Notice that, with the exception of the base variable `bar`, all other variable names differ when running without and with disambiguation. Thus, if you used a '`var-list-file`' generated on a run without the disambiguation information while running Kvasir with the disambiguation information, the names will not match up at all, and you will not get the proper selective variable tracing behavior.

The suggested way to use selective variable tracing with pointer type disambiguation is as follows:

1. First create the proper .disambig file by using either '`--disambig`' or '`--disambig-file`'. You can use '`--ppt-list-file`' as well to only create the .disambig file for certain program points, but do NOT use '`--var-list-file`' to try to create a .disambig only for certain variables; this feature does not work yet. Modify the variable entries in the Kvasir-generated .disambig file to suit your needs.
2. Now create a '`var-list-file`' by using '`--dump-var-file`' while running Kvasir with the .disambig file that you have just created. This ensures that the variables listed in '`var-list-file`' will have the proper names for use with that particular .disambig file. Modify the Kvasir-generated '`var-list-file`' to suit your needs.
3. Finally, run Kvasir with the '`--var-list-file`' option using the '`var-list-file`' that you have just created and either the '`--disambig`' or '`--disambig-file`' option with the proper .disambig file. This will perform the desired function: selective variable tracing along with disambiguation for all of the traced variables.

For maximum control of the output, you can use selective program point tracing, variable tracing, and disambiguation together all at once.

7.3.6 C++ support

Kvasir supports C++, but Kvasir has been tested more on C programs than on C++ programs, so Kvasir's C++ support is not as mature as its C support. Here is a partial list of C++ features that Kvasir currently supports:

- Class member functions are traced just like regular functions, except that their first parameter is a pointer (called `this`) to a single instance of the class. They are printed with the class name as the prefix, followed by a period and then the full function signature. For example, a `push()` function of a `Stack` class might be named `Stack.push(char*)`.

- OBJECT program points (see Section 5.2 [Program points], page 23) are printed out in the .decls file for each class with at least 1 member variable and 1 member function. No extra information besides member function traces are required in the .dtrace file; Daikon can link together class and function names to determine when a particular function is a member function and generate object invariants for that class by observing the values of the `this` parameter.
- Static member variables are currently treated just like global variables, because they actually have static global locations. Another (not yet implemented) possibility is to only print them at program points of member functions belonging to the respective variable's own class.
- Inheritance is handled correctly because whenever Kvasir traverses inside of a class to print out its member variables, it also recursively traverses inside all superclasses (and inside their superclasses, etc...) to print out inherited member variables. The superclass class names are appended onto the variable names to make them unique. For example, if `this` is an instance of a class that inherits from another class called `fooClass` which has a member variable `fooVar`, then Kvasir prints out `fooVar` as `this->fooClass.fooVar`. This correctly handles the case of multiple inheritance as well as several layers of inheritance. Thus, object invariants capture properties of a class's own member variables as well as those of its superclasses' member variables.
- Inheritance-based polymorphism is handled correctly without any extra effort because when a function entrance or exit is encountered at run time, the version that is called has already been resolved.
- Overloaded functions are handled correctly because Kvasir prints out the full function signature as its name in order to prevent conflicts. For example, two overloaded versions of a function `foo()` will be disambiguated by their signatures, such as `foo(int, int)` and `foo(double, double)`.
- Kvasir handles functions that pass parameters by reference as well as those that pass parameters by value.

One current C++ limitation is that Kvasir cannot print out the contents of classes which are defined in external libraries rather than in the user's program (e.g., it can properly output a C-string represented as `char*` but not the contents of the C++ `string` class). If further support for specific C++ features are important to you, please send email to `daikon-developers@lists.csail.mit.edu`, so that we can increase its priority on our to-do list.

7.3.7 Online execution

The term "online execution" refers to running Daikon at the same time as the target program, without writing any information to a file. This can avoid some I/O overhead, prevent filling up your disk with files, and in the future Daikon may be able to produce partial results as the target program is executing. A limitation of online execution is that, unless FIFOs, or named pipes (see Section 7.3.7.1 [Online execution with DynComp for C/C++], page 120) are used, it runs Daikon over only a single execution, as opposed to generalizing over multiple executions as can be done when writing to files and supplying all the files to Daikon.

The Chicory front end for Java also supports online execution via its '--daikon-online' argument (see Section 7.1.1.3 [Chicory miscellaneous options], page 92).

To use regular pipes in lieu of a disk file, simply use '-' as the name of the '.dtrace' file, and run the target program and Daikon in a Unix pipeline. When the '--dtrace-file=-' option is used to redirect the .dtrace output to stdout, the target program's stdout is redirected to the terminal ('/dev/tty') so that it does not intermix with the .dtrace output.

```
kvasir-dtrace --dtrace-file=- ./bzip2 --help | $DAIKON -
```

Of course, you could also replace '--help' with '-vv1 file.txt' to compress a text file (but start with a small one first).

(This example assumes that you have compiled the bzip2 example (in '$DAIKONDIR/examples/c-examples/bzip2' of the distribution) by saying `gcc -gdwarf-2 bzip2.c -o bzip2`, and that `$DAIKON` stands for the command that invokes Daikon, for instance `java -Xmx512m daikon.Daikon --config_option daikon.derive.Derivation.disable_derived_variables=true`.)

Instead of a regular pipe, you can use a named pipe, also known as a FIFO, which is a special kind of file supported by most Unix-compatible systems. When one process tries to open a FIFO for reading, it blocks, waiting for another process to open it for writing (or vice-versa). When both a reader and a writer are ready, the FIFO connects the reader to the writer like a regular Unix pipe.

The '--output-fifo' option causes Kvasir to create its output '.dtrace' file as a named pipe. When Kvasir is run with this option, Daikon needs to be run at the same time to read from the FIFO, such as from another terminal or using the shell's `&` operator.

For instance, the following two commands have the same effect as the pipeline above that used ordinary pipes. The FIFO is named 'bzip2.dtrace'.

```
kvasir-dtrace --output-fifo ./bzip2 --help &
$DAIKON bzip2.dtrace
```

The two commands (before and after the ampersand) could also be run in two different terminals.

7.3.7.1 Online execution with DynComp for C/C++

When running Kvasir with DynComp (using the '--with-dyncomp' option), Kvasir generates the .decls file after it generates the .dtrace file, so it is not possible to perform online execution using one run. The recommended way to perform online execution with DynComp is to run it once and only generate a .decls file with comparability information, then run Kvasir again without DynComp and pipe the .dtrace data directly into Daikon while using the .decls file generated from the previous run:

```
kvasir-dtrace --with-dyncomp --decls-only ./foo
```

This should generate a .decls file with comparability information named 'daikon-output/foo.decls'.

```
kvasir-dtrace --dtrace-no-decs --dtrace-file=- ./foo | java daikon.Daikon daikon-outpu
```

When you run Kvasir the second time, you don't need to run DynComp again since you are only interested in the .dtrace file. Notice that the .dtrace output is directed to standard out ('--dtrace-file=-') and does not contain any declarations ('--dtrace-no-decs') because the .decls file already contains the declarations. You can simply pipe that .dtrace output out to Daikon, which is invoked using the .decls file (with comparability information) generated during your previous run.

7.3.8 Installing Kvasir

The source code for Kvasir is included in the main Daikon distribution, and is compiled by default on Linux/x86 and Linux/x86-64 systems.

To compile and install Kvasir, give the command `make kvasir` from the top-level Daikon directory:

```
cd $DAIKONDIR
make kvasir
```

This will check that you have the appropriate prerequisites (such as GCC), configure Kvasir for your machine, compile it, and install it in the directory '`kvasir/inst`'.

You may see warnings about the 'missing' script. These can be ignored.

Once Kvasir has been installed, it can be used via the '`kvasir-dtrace`' script in the '`$DAIKONDIR/bin`' directory; if you have set up the Daikon environment according to the instructions above, it should already be in your `PATH`. For instructions on using Kvasir, see Section 7.3 [Kvasir], page 100.

7.3.9 Kvasir implementation and limitations

Kvasir is based on the Valgrind dynamic program supervision framework (which is best known for its memory error detection tool). Using Valgrind allows Kvasir to interrupt your program's execution, read its variables, and examine its memory usage, all transparently to the program. Also, rather than using your program's source code to find the names and types of functions and variables, Kvasir obtains them from debugging information included in the executable in a standard format (DWARF-2).

However, Kvasir has some limitations of its own. Because Kvasir uses Valgrind, it shares Valgrind's processor and operating system limitations. Furthermore, of the platforms supported by Valgrind, the only ones currently supported by Kvasir are `x86-linux` and `amd64-linux`. `x86-linux` refers to Intel 386-compatible processors (the so-called IA-32 architecture) such as the Intel Pentium and the AMD Athlon, running Linux. `amd64-linux` refers to the 64-bit extension of the x86 architecture found in many newer Intel and AMD processors, also variously referred to as x86-64, IA-32e, EM64T, and Intel 64, when running under a Linux kernel in 64-bit mode. The Itanium or IA-64 architecture is not supported. The Kvasir build process will automatically compile a 32-bit version, a 64-bit version, or both, whichever are supported by your system's default compiler.

Kvasir requires that your program have debugging information available in the DWARF-2 format, as produced by GCC version 3 and later. For the best results, the programs used by Kvasir should be compiled without optimization.

This subsection lists some of the known limitations of the current Kvasir release; if you encounter any problems other than listed here, please report them as bugs (see Section 9.5 [Reporting problems], page 162). The limitations are listed roughly in decreasing order of severity.

- Kvasir-traced programs take a while to start (often a good fraction of a second). When tracing short-lived programs, this overhead can dominate Kvasir's per-instruction runtime overhead. In order to make Kvasir run faster, try the '`--ignore-globals`' option in order to limit the amount of generated output. However, please keep in mind that, when running simultaneously with Daikon using the '`--output-fifo`' option (see Section 7.3.7 [Online execution], page 119), Kvasir can generate output data much faster than Daikon can process it. Thus, it is not the performance bottleneck in the entire invariant detection system.

- Kvasir's support for outputting arrays is not yet complete. It still does not have the functionality to print out multidimensional arrays with all of their elements or the option to flatten multidimensional arrays into multiple single-dimensional arrays.
- Kvasir behaves somewhat differently with different versions of GCC. We have had the best results with GCC versions 4.1 (which we use for testing) and 3.3/3.4. If feasible, we recommend that you use Kvasir with such a relatively recent version of GCC. Incompatibilities between Kvasir and the debugging information produced by older GCC versions can lead to incorrect output and, in some cases for version 2.95, can cause Kvasir to crash.
- Kvasir with Dyncomp will produce different results for x86 and x86-64 hosts. This is due to a Dyncomp limitation with regards to how handling the AMD64 ABI. The AMD64 ABI allows structs that are less than 8-bytes to be passed to a function via register. Dyncomp categorizes this as an interaction between all fields of the struct and will mark all fields of the struct as comparable to each other.
- Kvasir is incompatible with some compiler optimizations. It is definitely incompatible with the '`-fomit-frame-pointer`' optimization, and it may have trouble with other optimizations as well. We recommend that you compile programs for Kvasir without optimization.
- Kvasir always prints the contents of structures according to their compile-time type. Programs that use generic pointers and structural equivalence to simulate object-orientation will have derived-class fields missing when a structure is passed via a base-class pointer. This limitation can be worked around by manually coercing a pointer to a particular type (see Section 7.3.5.1 [Pointer type coercion], page 113).

7.4 Source-based C/C++ front end Mangel-Wurzel

In addition to the binary-based front end Kvasir (Section 7.3 [Kvasir], page 100), there is also a source-based C/C++ front end, Mangel-Wurzel. Unlike Kvasir, which runs only on Linux/x86 platforms, Mangel-Wurzel will run on nearly any Unix platform, as well as on Windows (with the Microsoft Visual C/C++ compiler). If you have access to a Linux/x86 platform (32 or 64-bit), we recommend use of Kvasir. Mangel-Wurzel is missing a few features of Kvasir; for instance, Mangel-Wurzel outputs fewer variables and has no abstract type inference analysis (). Mangel-Wurzel requires the source code for your program, and it also requires the Purify program analysis tool from Rational Software. You must install Purify (either a paid or evaluation version) before you can use Mangel-Wurzel.

Mangel-Wurzel is based on the EDG C/C++ front end, which is also commercial software. For this reason, Mangel-Wurzel is distributed in precompiled binary form only. See Section 7.4.6 [Installing Mangel-Wurzel], page 132, for more information about installation. If you have an EDG source license and would like to build or modify Mangel-Wurzel, we can provide source code on request.

Mangel-Wurzel is named after a root vegetable grown as cattle fodder. It "mangles" the source code into a form that is not fit for human consumption before compiling it.

7.4.1 Using Mangel-Wurzel

Mangel-Wurzel consists of three parts: a driver program called `mangel`, the preprocessor `wurzel` which adds annotations to the source code, and a small runtime library. Mangel is a replacement for the `cc` command: it produces an instrumented executable that behaves like the original program, but that also produces a '`.dtrace`' data trace file for Daikon to analyze. Mangel operates by in turn invoking the source preprocessor wurzel, the host compiler and linker, and the object code instrumenter Purify.

To use Mangel-Wurzel in conjunction with Daikon, follow these steps (after installing Mangel-Wurzel, see Section 7.4.6 [Installing Mangel-Wurzel], page 132):

1. Compile and link the program, using `mangel` in place of `cc`. This produces both an instrumented executable and also a '`.decls`' file for each compilation unit.

   If you use a makefile to compile your program, simply substitute `mangel` for `cc`.

   Alternately, you may issue the compilation commands directly. For example, suppose you have a program that consists of three input files, '`foo.c`', '`bar.c`', and '`baz.c`'. You can compile and link in one step:

   ```
   mangel -o foo foo.c bar.c baz.c
   ```

   Or, you can compile each file individually and then invoke mangel again to link:

   ```
   mangel -c foo.c
   mangel -c bar.c
   mangel -c baz.c
   mangel -o foo foo.o bar.o baz.o
   ```

   As a side-effect, the wurzel preprocessor invoked by mangel produces a '`.decls`' file for each source file it processes, in this case '`foo.decls`', '`bar.decls`', and '`baz.decls`'.

2. Run the instrumented program.

   - On Unix, simply run the instrumented program (in this example, '`foo`') created by mangel.
   - On Windows, Purify needs to be invoked when you run your program, not at link time. Mangel creates a '`.cmd`' file that does this for you; in this example, run '`foo.cmd`' instead of '`foo.exe`'. You can also run your program from within the Purify GUI instead of from the command line, if you prefer.

   This creates a single '`.dtrace`' file (in addition to anything else the program does). Assuming '`foo.c`' contains the `main` routine, the trace output for this example will be in '`foo.dtrace`'. How to run the instrumented program depends on your operating system.

3. Invoke Daikon, passing all of the '`.decls`' files and the '`.dtrace`' file as inputs.

   ```
   java daikon.Daikon foo.decls bar.decls baz.decls foo.dtrace
   ```

4. Examine the invariants. There are several ways to do this; See Section 3.1.1 [StackAr example], page 7.

7.4.2 Mangel options

Mangel acts as a replacement for the host `cc` command. It supports most of the common C compiler command-line options (see Section 7.4.2.1 [Standard compiler options for Mangel], page 124). It also has options that control how the wurzel preprocessor, compiler, and linker programs are invoked (see Section 7.4.2.2 [Mangel configuration options], page 125), and options that are specific to controlling the program annotations added by wurzel (see Section 7.4.2.3 [Mangel annotation options], page 127).

Mangel is invoked as:

```
mangel [options] inputfile1 ...
```

You can specify multiple *inputfiles* on the command line. Mangel will compile source files individually and then link them together (unless you have specified one of the command-line options that suppresses linking). You can also include object and archive files on the command line, which are passed directly to the linker.

7.4.2.1 Standard compiler options for Mangel

Mangel supports most of the common C compiler command-line options.

'`-o` *filename*'
: Specify *filename* as the output file.

'`-c`'
: Stop after compilation, producing a '`.o`' file; do not link.

'`-E`'
: Stop after preprocessing with wurzel, producing a '`.int.c`' file; do not compile or link. You might want to use this option for debugging if you are having problems getting the instrumented code to compile with the host C/C++ compiler.

'`-D`*symbol*'
'`-D`*symbol*`=`*value*'
: Define preprocessor symbols.

'`-U`*symbol*'
: Undefine preprocessor symbols.

'`-I`*pathname*'
: Add *pathname* to the list of places searched for include files.

If the environment variable `MANGEL_DIR` is set, mangel implicitly adds its '`include`' subdirectory to the list of directories searched for system include files. Otherwise you must explicitly use '`-I`' to specify where to find Mangel-Wurzel's system include files, or include this information in the options list passed directly to the wurzel preprocessor. See the '`--preprocessor_opts`' option (see Section 7.4.2.2 [Mangel configuration options], page 125).

On Windows, mangel implicitly adds '`-I`' options for all of the pathnames specified in the environment variable `INCLUDE`, which is also used by the Microsoft C compiler to specify the location of its default system libraries. On Unix, it uses the path specified in the environment variable `CPATH` in addition to the default system libraries specified in the configuration file, for compatibility with gcc.

'`-O`'
'`-O`*value*'
: Enable optimization in the compiler. This might interact badly with Purify, so be careful.

'-g' Enable debugging in the compiler. Since Purify expects code it processes to be compiled with '-g', this is typically configured as one of the default host compiler options (see Section 7.4.6 [Installing Mangel-Wurzel], page 132) and need not be supplied explicitly on the command line.

'-L*pathname*'
: Add *pathname* to the list of places searched for system libraries. This option is not supported on Windows, since the Microsoft linker uses a different model for finding libraries.

'-l*name*' Link with system library *name*. This option is not supported on Windows, since the Microsoft linker expects libraries to be specified like ordinary input files to the linker.

If the environment variable `MANGEL_DIR` is set, mangel implicitly adds '`$MANGEL_DIR/lib/libmangelwurzel.a`' to the list of libraries to be linked with. Otherwise you must explicitly link with this library, include this information in the options list passed directly to the linker. See '`--linker_opts`' (see Section 7.4.2.2 [Mangel configuration options], page 125).

Also, on Unix, if the environment variable `PURIFY_DIR` is set, mangel implicitly adds '`$PURIFY_DIR/purify_stubs.a`' to the list of libraries to be linked with. Similarly, if the environment variable is not set, you have to specify this library explicitly. This is not needed on Windows because Purify uses a different model for resolving link references to its API on that platform.

'--c'
'--c99'
'--c++' Interpret the input source file as old standard C (C89), C99, or C++, respectively. If none of these options are specified, mangel attempts to infer the source language from the filename extension.

'--ansi'
'--gcc'
'--microsoft'
: Enable strict ANSI, gcc, and Microsoft compatibility modes (respectively) in the wurzel preprocessor. If none of these options are specified, wurzel accepts ANSI C/C++ with a few common extensions and anachronisms.

'-h'
'--help' Print a usage message and exit.

7.4.2.2 Mangel configuration options

You can use either command-line options or an options file to override Mangel's default behavior regarding how Mangel invokes wurzel, the host compiler, and the linker. Options files are discussed in Section 7.4.2.4 [Options files for Mangel], page 128.

When you use command-line options from the shell or makefile to specify these configuration parameters, be careful to quote the entire option value string so that mangel interprets it as a single argument.

'`--preprocessor` *`command`*'
: Use *command* to invoke the wurzel preprocessor. If this option is not specified on the command line or in an options file, it defaults to '`wurzel`'.

'`--preprocessor_opts` *`option_string`*'
: Use *option_string* as additional options to the wurzel preprocessor. If this option is not specified on the command line or in an options file, it defaults to a system-specific value.

'`--compiler` *`command`*'
: Use *command* to invoke the host C/C++ compiler. If this option is not specified on the command line or in an options file, it defaults to '`cc`'.

'`--compiler_opts` *`option_string`*'
: Use *option_string* as additional options to the host compiler. If this option is not specified on the command line or in an options file, it defaults to a system-specific value.

'`--linker` *`command`*'
: Use *command* to invoke the host linker. If this option is not specified on the command line or in an options file, it defaults to '`cc`'.

'`--linker_opts` *`option_string`*'
: Use *option_string* as additional options to the host linker. If this option is not specified on the command line or in an options file, it defaults to a system-specific value.

'`--purify` *`command`*'
: Use *command* to invoke Rational Purify. If this option is not specified on the command line or in an options file, it defaults to '`purify`'. See Section 7.4.5 [Interaction with Purify], page 131, for more information about Purify options.

'`--purify-opts` *`option-string`*'
: Use *option-string* as additional options to Rational Purify. If this option is not specified on the command line, it defaults to a system-specific value. See Section 7.4.5 [Interaction with Purify], page 131, for more information about Purify options.

'`--tmpdir` *`pathname`*'
: Put temporary files (the annotated '`.int.c`' files and temporary object files created prior to linking) in the indicated directory. If this option is not specified on the command line or in an options file, it defaults to a system-specific default, '`/tmp/`' on Unix platforms or the directory specified by the `TMP` or `TEMP` environment variables on Windows. The directory must exist and be writable.

'`-v`'
'`--verbose`'
: Print commands (to `stderr`) before executing them.

'`-n`'
: Print commands (to `stderr`), but don't execute them.

7.4.2.3 Mangel annotation options

Mangel passes these options to the wurzel preprocessor, which adds instrumentation to the code and produces a '`.decls`' file for each source file.

'`--decls-file` *`filename`*'
: Specify *filename* as the name of the '`.decls`' file for this compilation unit. If this option is not specified, declarations are written to a file in the current directory with a name derived from the current compilation unit.

'`--dtrace-file` *`filename`*'
: Specify *filename* as the name of the '`.dtrace`' file for this program. This option is only used if the compilation unit contains a definition for `main`. If this option is not specified, trace output is written to a file in the current directory when the executable is run, with a name derived from the file containing the definition of `main`.

  If you specify a *filename* of '`-`', trace output is written to stdout. In this case, regular program output to stdout is redirected to the terminal (on Unix) or console (on Windows). This allows you to redirect trace output to a file or pipe on the command line, without mingling the two output streams.

'`--dtrace-append`'
: Specify that output should be appended to an existing '`.dtrace`' file, rather than overwriting it. This option is only used if the compilation unit contains a definition for `main`.

'`--dtrace-gzip`'
: Produce gzipped (compressed) trace output. This is implemented by filtering the trace output through `gzip`. This option is only used if the compilation unit contains a definition for `main`.

  The runtime library invokes gzip as a subprocess, via a pipe. If you have gzip installed in a nonstandard location on your system, you can specify its pathname using the `GZIP` environment variable.

'`--ignore-globals`'
: Ignore global, file-scope static, and function-scope static variables when emitting program point state. The default behavior is to emit declarations and trace information for static variables and global variables that are defined (not just declared as `extern`) in the compilation unit and are in scope at the given program point, as well as (for C++ class member functions) static member variables of the containing class.

'`--ignore-static-vars`'
: Ignore file-scope and function-scope static variables when emitting program point state, but include global variables defined in the compilation unit.

'`--nesting-depth` *`depth`*'
: Specify the recursion depth for examining fields of nested `struct`, `union`, and `class` types. The default value is 2.

'`--flatten-arrays`'
: This option forces the flattening of statically-sized arrays into separate variables, one for each element. For example, an array *foo* of size 3 would be flattened into 3 variables: *foo[0]*, *foo[1]*, *foo[2]*. By default, Mangel-Wurzel flattens arrays only after is has already exhausted the one level of sequences that Daikon allows in the .dtrace output format.

'`--max-flatten-array-size` *size*'
: Specify the maximum *size* of any dimension for arrays to be flattened. The default value is 10, so that (for instance) a 10x10x20 array would be broken down into 100 variables each containing a 20-element sequence. If any one dimension to be flattened is larger than *size*, then Mangel-Wurzel doesn't output the contents of the array.

'`--max-array-size` *size*'
: Specify the maximum *size* of one-dimensional arrays, which are output as sequences in the .dtrace file. If the static array dimension is larger than *size*, then Mangel-Wurzel doesn't output the contents of the array. The default is 100.

'`--ignore-system-structs`'
: If you specify this option, Mangel-Wurzel ignores fields of `struct`, `union`, and `class` types declared in system include files (e.g., those included via the '`#include <file.h>`' syntax rather than '`#include "file.h"`'.) These are typically interfaces like `FILE` whose internal representations are not interesting to "regular" programmers or portable across operating systems.

'`--ppt-select-pattern` *regex*'
: Only emit program points that match *regex*. Specifically, a program point is considered to match *regex* if any of the function name, program point name, or containing class name (for C++) match *regex*.

  This option can be specified multiple times, but cannot be used in conjunction with '`--ppt-omit-pattern`'.

'`--ppt-omit-pattern` *regex*'
: Suppress program points that match *regex*. Specifically, a program point is considered to match *regex* if any of the function name, program point name, or containing class name (for C++) match *regex*.

  This option can be specified multiple times, but cannot be used in conjunction with '`--ppt-select-pattern`'.

7.4.2.4 Options files for Mangel

In addition to specifying mangel options on the command line, you can also put them in an options file. This is most useful for customizing the defaults for the configuration options (Section 7.4.2.2 [Mangel configuration options], page 125) which control how wurzel and the host compiler and linker should be invoked, but you can put any command-line options in the file.

Mangel looks for an options file in the following places, and will use the first one it finds.

1. '`.mangelrc`'
2. '`$HOME/.mangelrc`'

3. '$MANGEL_DIR/.mangelrc'

Option files are formatted with one option, or option/value pair, per line. Everything following the option keyword on the same line is considered to be a value. Value strings should not be quoted (even if they contain embedded spaces). Blank lines are ignored.

Here is an example of an options file:

```
--preprocessor wurzel
--preprocessor_opts --sys_include=/usr/include

--compiler cc
--compiler_opts -g -Dsetjmp=_setjmp -Dva_copy=__va_copy

--linker cc
--linker_opts -lstdc++

--purify purify
--purify_opts -log-file=/dev/null -append-logfile
```

7.4.3 Pointer/array disambiguation in Mangel-Wurzel

Wurzel currently does not support any form of pointer/array disambiguation. In other words, an object of type '`int *`' is treated as a pointer to a single integer, not as the address of an array of `int`. This is in contrast to the default behavior of Kvasir (Section 7.3 [Kvasir], page 100), which assumes the more general case that pointers point to arrays rather than single objects. The difference in behavior is due to differences in the underlying runtime support; Purify does not track the length of arrays or provide an API for querying this information.

Wurzel does, however, make use of static type information to emit trace information for function arguments and global and static variables with fully-defined array types as arrays. Note that a "fully-defined" array type includes a complete set of (constant) dimensions.

Wurzel also assumes that variables with type `char *`, or arrays of `char`, are likely to be null-terminated strings rather than pointers to a single `char` object. The runtime code which prints strings first performs a "sanity check" to make sure the string is of reasonable length and contains only printable characters (as determined by the C library functions `isprint` and `isspace`) rather than arbitrary byte values. If these conditions are not met, the string is printed as its first character followed by '`...`' (such as '`"a..."`'), or as "nonsensical" if the first character is not printable.

On the other hand, variables with type `signed char *` or `unsigned char *`, or arrays of explicitly `signed` or `unsigned char`, are treated as byte pointers or byte arrays, respectively, and the elements print as normal integer values rather than as strings. Multidimensional `char` arrays also print as byte arrays rather than as arrays of strings.

7.4.4 Mangel-Wurzel usage notes

You do not need to instrument all compilation units in a program with Mangel-Wurzel if you are only interested in analyzing certain parts of it with Daikon; you can simply compile the "uninteresting" files in the normal way with your usual C compiler. However, you must either instrument the compilation unit including the definition of `main` so that the necessary runtime initialization is performed when the program starts, or insert an explicit call to `mw_init` in your program before any of the instrumented code is executed. You should also use mangel to link your program.

The wurzel preprocessor produces the same dialect of code that it receives as input, that is C, C99, or C++. It does not translate C++ to C. This ensures that preprocessed code is link-compatible with system C++ libraries and other code that is not processed with wurzel.

While wurzel recognizes standard C++, its support of instrumentation for C++ programs is rather rudimentary at this point. In particular, it ignores anything having to do with templates. Wurzel is smart enough to only generate references to class fields that are accessible at a given program point, to avoid errors in the compilation phase, but at this time it will not also generate references to accessible base classes and their members. (This should be optional in any case because it will greatly increase the number of variables at a given program point.)

If you are using the C++ libraries that came with your host C++ compiler, you will probably have to add the locations of the include files to the search path passed to wurzel, as well adding the libraries themselves to your link options.

If your source code uses extensions specific to a particular host compiler, you may need to use the '`--gcc`' or '`--microsoft`' flags to enable the appropriate compatibility mode. Wurzel recognizes many other command-line options supported by the underlying EDG front end to customize the language dialects it accepts, as described in the EDG documentation. You can set these options using the '`--preprocessor-opts`' flag to mangel.

If your code compiles with your host compiler, but not with Mangel-Wurzel, you can try these approaches:

1. If you are getting compilation errors from wurzel that can't be resolved with one of the above compatibility flags or with appropriate defines, a possible workaround is to preprocess the code with the host compiler's preprocessor before running it through Mangel-Wurzel.
2. If you are getting compilation errors from the host C compiler, you can use the '`-E`' option to mangel to produce a '`.int.c`' file containing the code as preprocessed by wurzel, which you can examine to track down the cause of the errors.

Note that since the output of wurzel is compilable source code, it is possible to perform the preprocessing on one machine and then compile, link, and run on another platform supported by Rational Purify.

7.4.4.1 Using Mangel-Wurzel on Unix

GCC is the standard compiler on most Unix-based systems. You may find it necessary to specify the '`--gcc`' option to wurzel to make it correctly process system include files, even if your code doesn't use GCC extensions itself.

Not all of the preprocessor symbols predefined by GCC's cpp are correctly predefined by the EDG front end's GCC compatibility mode, so you may need to specify some of these preprocessor symbols yourself. If you run into such preprocessing errors, consider creating a '`mwdefines.h`' file instead of trying to define every necessary macro on the command line for wurzel:

1. Create a file with a list of all of cpp's built-in defines: '`cpp -dM > mwdefines.h`'

2. Edit the file, wrapping each of the '`#define`'s in an '`#ifndef...#endif`' to prevent multiple definition.
3. In the '`--preprocessor-opts`' value in your '`.mangelrc`', add '`--preinclude=filename`', where *filename* is the name of your new '`mwdefines.h`' file.

7.4.4.2 Using Mangel-Wurzel on Windows

Rational Purify on Windows only supports the Microsoft Visual C/C++ compiler, not GCC. Therefore Mangel-Wurzel will only work with the Microsoft compiler on this platform as well. Currently, Mangel-Wurzel is not integrated with the Visual Studio IDE, and runs as a command-line utility only.

You can run Mangel-Wurzel and programs instrumented with Mangel-Wurzel from within a Cygwin shell, Makefile, and the like, as well as from the default Windows '`cmd`' shell in a console window. However, another limitation of Rational Purify on Windows is that instrumented executables will only run from a console window, not an XTerm or other Cygwin pty. (This is because Purify, like some other Windows console programs, does not like having its standard input or output redirected to pipes.)

Since Mangel-Wurzel runs as a regular Windows console application, it expects all pathname arguments and environment variables to be in regular Windows pathname syntax even when you run it from inside a Cygwin shell.

7.4.5 Interaction with Purify

If you are interested in obtaining Purify diagnostics for your code, you should run Purify on the uninstrumented program. The remainder of this section explains why this is the case. It is not necessary to fix all the problems that Purify diagnoses before running Mangel-Wurzel. However, fixing them is likely to correct bugs and make your code more robust, so it is recommended.

Mangel-Wurzel's runtime library relies on the Purify API to test the validity of pointers — in particular, to check for pointers to unallocated memory and pointers to memory that has been allocated but not initialized. It performs these checks so that it can avoid dereferencing bad pointers and uninitialized variables when printing values at program points to the '`.dtrace`' file. (Instead, it prints a value of "nonsensical" for those variables, which has a special meaning to Daikon (see section "Nonsensical values" in *Daikon Developer Manual*).)

Both the Unix and Windows versions of Purify have problems identifying uninitialized variables on the stack. In Mangel-Wurzel, this manifests itself by uninitialized block-scope automatic variables sometimes printing as garbage values rather than as "nonsensical", which in turn might cause Daikon to find inaccurate invariants.

Normally, mangel is configured to tell Purify to discard all of its diagnostic messages. This is because, due to limitations of the Purify API, it isn't possible for the Mangel-Wurzel runtime to run Purify completely silently, and the diagnostics will contain bogus messages generated by the Mangel-Wurzel runtime as well as genuine diagnostics about the program that has been instrumented.

Specifically, since the Unix versions of Purify don't provide any API functions for checking the validity of pointers without generating messages, the Mangel-Wurzel runtime works around this by enabling message batching and by discarding the batched messages as processing of each program point is completed.

7.4.6 Installing Mangel-Wurzel

Since Mangel-Wurzel is based on the proprietary EDG C/C++ front end, we distribute it, at `http://pag.csail.mit.edu/daikon/download/`, in binary rather than source format. (If the web page does not currently contain a binary for your architecture, send mail to `daikon-developers@lists.csail.mit.edu` and we may be able to produce one for you.) The distribution does not include Purify, which you must install separately (either purchase it or download an evaluation copy).

Installation requires unpacking the distribution, setting the `MANGEL_DIR` and `PURIFY_DIR` environment variables, optionally customizing the defaults for how mangel operates, and optionally running its tests.

1. Download the archive from `http://pag.csail.mit.edu/daikon/download/` and unpack it in the parent of the directory where you wish to install Mangel-Wurzel. Unpacking creates a directory named '`mangel-wurzel`'.
   - On the Linux/x86 platform, execute these two commands (other Unix platforms are similar):
     ```
     wget http://pag.csail.mit.edu/daikon/download/binaries/mangel-wurzel-linux-x86.tar.gz
     tar xvzf mangel-wurzel-linux-x86.tar.gz
     ```
   - On Windows, download, then unzip, `http://pag.csail.mit.edu/daikon/download/binaries/mang`
     .
2. Set these environment variables:

   `MANGEL_DIR`
   Set to the full pathname of the '`mangel-wurzel`' directory.

   `PATH` Add '*`MANGEL_DIR`*`/bin`' to your path.

   `PURIFY_DIR`
   Set to the directory in your Purify distribution containing '`libpurify_stubs.a`'.
3. Customize the default commands and options mangel uses to invoke the wurzel preprocessor, the host compiler and linker, and Purify. Make these changes in the system options file '`mangel-wurzel/.mangelrc`'; see Section 7.4.2.4 [Options files for Mangel], page 128 for details about the format of this file.

   A suggested version of '`.mangelrc`' for each supported platform is included with the distribution. For example, '`.mangelrc.linux`' is provided for Linux. You should copy the appropriate file to '`.mangelrc`' and make any necessary changes there. The most likely things you may need to customize are:
   - Make sure that the '`--compiler`' and '`--linker`' commands are correct. For example, if you want to use gcc, on some systems you may need to specify '`gcc`' explicitly instead of '`cc`', or use a full pathname. On Windows, you should use '`cl`' only.
   - You may need to alter the default system include paths, specified with '`--sys_include`' in the '`--preprocessor_opts`' option. See Section 7.4.4.1

[Using Mangel-Wurzel on Unix], page 130, and Section 7.4.4.2 [Using Mangel-Wurzel on Windows], page 131, for details.

- On Unix, you must specify the full set of search paths for include files in your '`.mangelrc`' file, since these are normally hard-wired into GCC instead of configured with an environment variable. These include your gcc include directory (run '`gcc -print-libgcc-file-name | sed s/libgcc.a/include/`'), '`/usr/local/include`', and '`/usr/include`' (after the gcc include directory).
- On Windows, Mangel-Wurzel uses the same `INCLUDE` environment variable as the Microsoft compiler to set up the system include paths, so in theory you should not have to specify any additional paths in your '`.mangelrc`' configuration file. Do be sure that you specify '`cl`' (the Microsoft C compiler) for both the '`--compiler`' and '`--linker`' options and that you have your `PATH` environment variable set up to include both the path to the cl compiler and related Visual C utilities, and the path for Rational Purify.

  Make sure you set up your '`.mangelrc`' to compile with debugging information enabled; otherwise Rational Purify does not detect uninitialized memory references, which is a critical feature for Mangel-Wurzel's data structure traversal. On Windows, Purify also has problems reliably detecting uninitialized stack locations which manifest themselves in Mangel-Wurzel as uninitialized block-scope automatic variables sometimes being reported as having garbage values instead of as being "nonsensical", which is treated specially by Daikon (see section "Nonsensical values" in *Daikon Developer Manual*).

  You will probably need to specify the '`--microsoft`' compatibility mode option as the default in your '`.mangelrc`' file.

- You might also want to add some additional options, such as specifying '`--gcc`' or '`--microsoft`' to enable the appropriate compatibility mode for your normal C compiler by default.

For debugging the options file, you can use the '`-n`' or '`-v`' options to see the actual commands mangel is generating to invoke each phase.

4. Optionally (but recommended), run Mangel-Wurzel's tests.

```
cd tests
make
```

7.5 Perl front end dfepl

This section contains details about dfepl, the Daikon front end for Perl. For a brief introduction to dfepl, see Section 3.3.2 [Perl examples], page 14 and Section 3.3.1 [Instrumenting Perl programs], page 14.

dfepl works with Perl versions 5.8 and later. (To be precise, Perl programs instrumented with dfepl can also be run with Perl 5.6, but the instrumentation engine, which is itself written in Perl, requires version 5.8). dfepl reads the source code for Perl modules or programs, and writes out instrumented versions of that code that keep track of function parameters, and make calls to routines in the '`daikon_runtime`' package whenever an instrumented subroutine is entered or exited.

The instrumentation engine recognizes parameters as those variables that are declared with `my(...)` or `local(...)` and, in the same expression, assigned to from a value related to the argument array `@_`, but only among the first contiguous series of such assignments in the body of a subroutine. This will capture the most common assignment idioms, such as `my $self = shift;` (where `shift` is short for `shift @_`), `my $x = $_[0];`, and `my($x, $y, @a) = @_;`, but the arguments to subroutines which access them only directly through `@_`, or that perform other operations before reading their arguments, will not be recognized.

If the uninstrumented code requested warnings via the `use warnings` pragma or by adding the '`-w`' flag on the `#!` line, the instrumented code will also request warnings. In this case, or if '`-w`' is specified on the command line when running it, the instrumented code may produce warnings that the original code did not. There are several situations in which the instrumented code produced by dfepl, while functionally equivalent to the original, generates more warnings. The most common such problem, which arises from code that captures the scalar-context return value of a subroutine that returns a list, has been avoided in the current version by disabling the warning in question. Other warnings which are known to be produced innocuously in this way include '`Ambiguous call resolved as CORE::foo(), qualify as such or use &`' (caused by code that uses `CORE::` to distinguish a built-in function from a user subroutine of the same name), and '`Constant subroutine foo redefined`' (caused by loading both instrumented and uninstrumented versions of a file). Though some such warnings represent deficiencies in the instrumentation engine, they can be safely ignored when they occur.

Because Perl programs do not contain static type information to distinguish, for instance, between strings and numbers, the Perl front end incorporates an additional dynamic analysis to infer these types. This type guessing, which occurs as a first pass before the program can be instrumented to produce output for Daikon, operates in a manner somewhat analogous to Daikon itself: watching the execution of a program, the runtime system chooses the most restrictive type for a variable that is not contradicted during that execution. These types indicate, for instance, whether a scalar value always holds an integer, a possibly fractional numeric value, or a reference to another object. It should not be necessary to examine or modify this type information directly, but for the curious, the syntax of the type information is described in comments in the '`Daikon::PerlType`' module.

The safest course is to infer types for variables using exactly the same program executions (e.g., test cases) which will later be used to generate traces for Daikon, as this guarantees that the type information will match the actual data written to the trace file. However, because the type-guessing-instrumented versions of programs run fairly slowly in the current version, you may be tempted to use a subset of the input data for type guessing. Doing so is possible, but it will only work correctly if the smaller tests exercise all of the instrumented subroutines and exit points with all the types of data they will later be used with. If the trace runtime tries to output a data value that doesn't match the inferred type, the value may silently be converted according to Perl's usual conventions (for instance, a non-numeric string may be treated as the number zero), or it may cause an error during tracing (for instance, trying to dereference a supposed array reference that isn't). Also, if a subroutine exit point is traced but was never encountered during type guessing, the generated '`.decls`' and '`.dtrace`' files will be incompatible in a way that will cause Daikon to abort with an error message of the form '`Program point foo():::EXIT22 appears in dtrace file but not in any decl file`'.

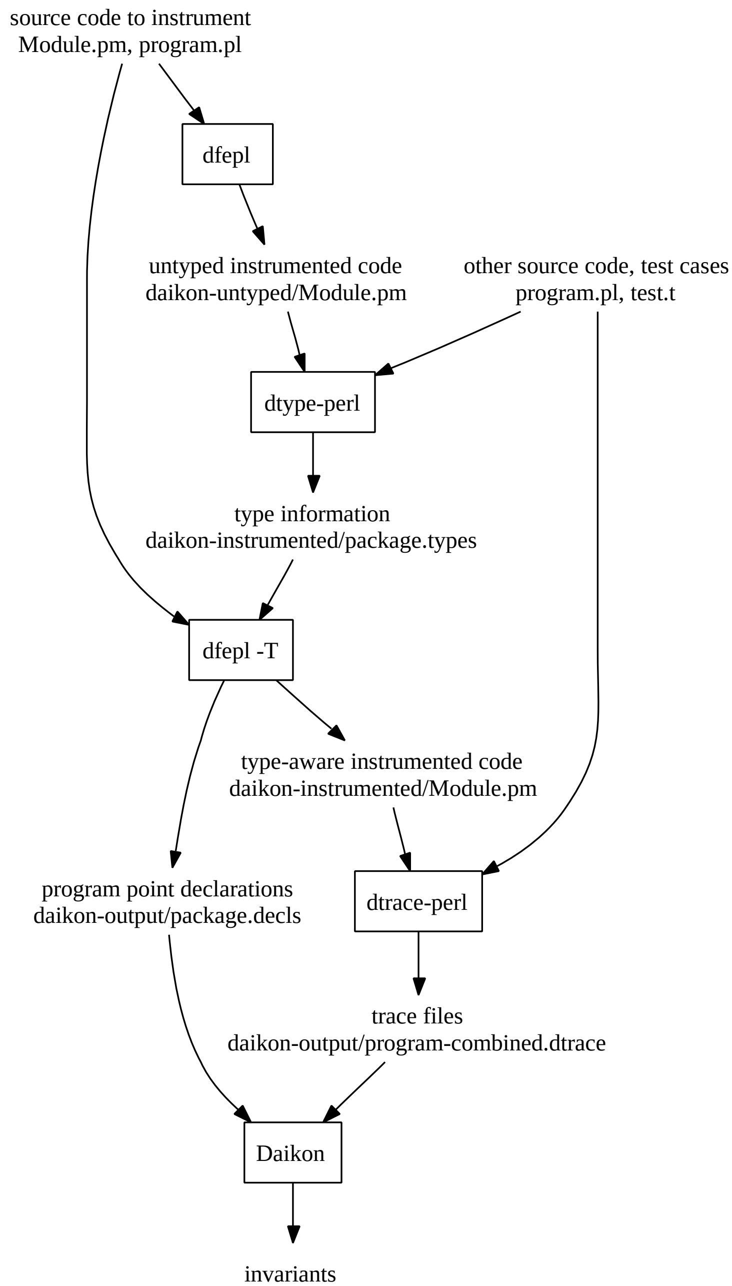

Figure 7.1: Workflow of instrumenting Perl code with dfepl.

dfepl works by reading one or more Perl programs or modules, and writing out new versions of those files, instrumented to capture information about their execution, by default to another directory. dfepl is used in two passes: first, before type information is available, instrumented versions are written to a directory '`daikon-untyped`'. These untyped programs, when run, will write files containing dynamically inferred type information (with the extension '`.types`'), by default to the '`daikon-instrumented`' directory. When dfepl is rerun with this type information, it produces type-aware instrumented code in the '`daikon-instrumented`' directory, which when run produces execution traces in files with the extension '`.dtrace`' in the a directory '`daikon-output`'.

7.5.1 dfepl options

'`--absolute`'
'`--no-absolute`'

> '`--absolute`' stores the absolute path to the output directories (by default named '`daikon-untyped`', '`daikon-instrumented`' or '`daikon-output`') in the instrumented programs, so that no matter where the instrumented program is run, the output will go to a fixed location. Even if these directories are given as relative paths (as is the default), '`--absolute`' specifies that they should always be taken as relative to the directory that was the working directory when dfepl was run.
>
> '`--no-absolute`' specifies the opposite, causing the output paths to be interpreted relative to the current working directory each time the instrumented program is invoked. The default, when neither option is specified, is for '`.types`' files to use an absolute path, but all others to use relative path, so that the '`.types`' files will always be in the same place as the instrumented source files that generated them, but the '`daikon-output`' directory will be created in the current directory when the program runs.

'`--accessor-depth=`*num*'

> Controls the number of nested invocations of object accessor methods to examine. For instance, suppose that the `Person` class has a method `mother()` that returns another person (and has been specified to dfepl as an accessor), and that `$me` is an instrumented variable. If the accessor depth is 1, only `$me->mother()` will be examined. If the depth is 2, `$me->mother()->mother()` will also be examined. Specifying large accessor depths is generally not advisable, especially with many accessor methods, as the number of variables examined can be too many for Daikon to process efficiently.
>
> By default, the Daikon Perl trace runtime will examine at most a single level of accessors.

'`-A`'
'`--accessors-dir=`*directory*'

> Look for files containing accessor lists in *directory*, or the current directory if *directory* is omitted. For a class `Acme::Foo`, accessors are methods that return information about an object but do not modify it. dfepl cannot determine on its own which methods are accessors, but when a list of them is provided, it can call an object's accessors when examining a variable of that class to obtain more information about the object. To tell dfepl about the accessors for `Acme::Foo`, make a file listing the names of each accessor method, one per line with no other punctuation, named '`Acme/Foo.accessors`' in the same directory as '`Acme/Foo.pm`'.

'`--decls-dir=`*`directory`*'
: Put generated declaration files in *directory* and its subdirectories. The default is '`daikon-output`'.

'`--decls-style=`*`style`*'
: *style* should be one of '`combined`', '`flat`', or '`tree`'. A style of '`combined`' specifies that the declarations for all packages should be merged, in a file named '`prog-combined.decls`' where '`prog`' is the name of the program. A style of '`flat`' specifies that the declarations for each package should be in a separate file named after the package, but that these files should go in a single directory; for instance, the declarations for `Acme::Trampoline` and `Acme::Skates::Rocket` would go in files named '`Acme::Trampoline.decls`' and '`Acme::Skates::Rocket.decls`'. A style of '`tree`' specifies that each package should have its own declarations file, and that those files should be arranged in directories whose structure matches the structure of their package names; in the example above, the files would be '`Acme/Trampoline.decls`' and '`Acme/Skates/Rocket.decls`'.

  The default is '`tree`'. Note that '`--decls-style`' and '`--types-style`' are currently constrained to be the same; if one is specified, the other will use the same value.

'`--dtrace-append`'
'`--no-dtrace-append`'
: When '`--dtrace-append`' is specified, the instrumented program will append trace information to the appropriate '`.dtrace`' file each time it runs. When '`--no-dtrace-append`' is specified, it will overwrite the file instead.

  The default behavior is to overwrite. This choice can also be overridden, when the program is run, to always append by setting the environment variable `DTRACEAPPEND` to 1.

  When appending to a '`.dtrace`' file, no declaration information is ever produced, because it would be redundant to do so and Daikon does not permit re-declarations of program points.

'`--dtrace-dir=`*`directory`*'
: Put generated trace files in *directory* and its subdirectories. The default is '`daikon-output`'.

'`--dtrace-style=`*`style`*'
: *style* should be one of '`combined`', '`flat`', or '`tree`'. A style of '`combined`' specifies that the traces for all packages should be merged, in a file named '`prog-combined.dtrace`', where '`prog`' is the name of the program. A style of '`flat`' specifies that the traces for each package should be in a separate file named after the package, but that these files should go in a single directory; for instance, the declarations for `Acme::Trampoline` and `Acme::Skates::Rocket` would go in files named '`Acme::Trampoline.dtrace`' and '`Acme::Skates::Rocket.dtrace`'. A style of '`tree`' specifies that each package should have its own trace file, and that those files should be arranged in directories whose structure matches the structure of their package names;

in the example above, the files would be '`Acme/Trampoline.dtrace`' and '`Acme/Skates/Rocket.dtrace`'.

The default is '`combined`'.

'`--help`' Print a short options summary.

'`--instr-dir=`*`directory`*'
'`--instrsourcedir=`*`directory`*'
: Put instrumented source files in *directory* and its subdirectories. The default is '`daikon-untyped`', or '`daikon-instrumented`' if type information is available.

'`--list-depth=`*`DEPTH`*'
: Consider as many as *DEPTH* of the first elements of a list to be distinct entities, for the purpose of guessing their types. When subroutines return a list of values, each value may have a distinct meaning, or the list may be homogeneous. When trying to assign types to the elements of a list, the Daikon Perl trace runtime will try making separate guesses about the types of the elements of a short list, but it would be inefficient to make retain this distinction for many elements. This parameter controls how many elements of a list will be examined individually; all the others will be treated uniformly.

  The default is 3.

'`--output-dir=`*`directory`*'
: Put all of the files that are the output of the tracing process (and therefore input to the Daikon invariant detection engine) in *directory* and its subdirectories. This option is a shorthand equivalent to setting both '`--decls-dir`' and '`--dtrace-dir`' to the same value.

  The default behavior is as if '`--output-dir=daikon-output`' had been specified.

'`--perl=`*`path`*'
: Use *path* as the location of Perl when calling the annotation back end (a module named `B::DeparseDaikon`), rather than the version of Perl under which `dfepl` itself is running, which is probably the first `perl` that occurs on your path. For instance, if the first version of `perl` on your path isn't version 5.8 or later, you should this option to specify another `perl` program that is.

'`--nesting-depth=`*`num`*'
: When examining nested data structures, traverse as many as *num* nested references. For instance, suppose that `@a` is the array

```
@a = ({1 => [2, 3]}, {5 => [4, 2]})
```

  If the depth is 0, then when examining `@a`, Daikon's Perl trace runtime will consider it to be an array whose elements are references, but it won't examine what those references point to. If the depth is 1, it will consider it to be an array of references to hashes whose keys are integers and whose values are references, but it won't examine what *those* references point to. Finally, if the depth is 2 or more, it will consider `@a` to be an array of references to hashes whose keys are integers and whose values are references to arrays of integers.

  The default nesting depth is 3.

When referenced objects have accessor methods, or when accessors return references, the '`--accessor-depth`' and '`--nesting-depth`' options interact. Specifically, if these depths are A and R, the behavior is as if the runtime has a budget of 1 unit, which it can use either on accessors which cost 1/A or references which cost 1/R. It may thus sometimes be useful to specify fractional values for '`--accessor-depth`' and '`--nesting-depth`'; in fact, the default accessor depth is 1.5.

'`--types-append`'
'`--no-types-append`'
: When '`--types-append`' is specified, the instrumented program will append type information to the appropriate '`.types`' file each time it runs. When '`--no-types-append`' is specified, it will overwrite the file instead.

The default behavior is to append. If '`--no-types-append`' is specified, however, this choice can also be overridden, when the program is run, to append by setting the environment variable `TYPESAPPEND` to 1. There is no way to use environment variables to force the runtime to overwrite a types file, but an equivalent effect can be obtained by simply removing the previous types file before each run.

'`-T`'
'`--types-dir=`*`directory`*'
: Look for '`.types`' files in *directory*, or '`daikon-instrumented`' if *directory* is omitted. When instrumenting a module `Acme::Trampoline`, used in a program '`coyote.pl`', dfepl will look for files named '`coyote-combined.types`', '`Acme::Trampoline.types`', and '`Acme/Trampoline.types`', corresponding to the possible choices of '`--types-style`'. Once discovered, the files are used in the same way as for '`-t`'.

'`--types-file=`*`file`*'
'`-t` *`file`*'
: Include type information from *file* when instrumenting programs or modules. Since Daikon needs to know the types of variables when they are declared, useful '`.decls`' and '`.dtrace`' files can only be produced by source code instrumented with type information. Since Perl programs don't include this information to begin with, and it would be cumbersome to produce by hand, type information must usually be produced by running a version of the program that has itself been annotated, but without type information. The Daikon Perl trace runtime will automatically decide whether to output types, or declarations and traces, depending on whether the source was instrumented without or with types. This option may occur multiple times, to read information from multiple types files (irrelevant type information will be ignored).

'`--types-basedir=`*`directory`*'
: Put files containing type information in *directory* and its subdirectories. By default, this is whatever '`--instr-dir`' is, usually '`daikon-instrumented`'.

'`--types-style=`*`style`*'
: *style* should be one of '`combined`', '`flat`', or '`tree`'. A style of '`combined`' specifies that the types for all packages should be merged, in a file named '`prog-combined.types`', where '`prog`' is the name of the program. A

style of 'flat' specifies that the types for each package should be in a separate file named after the package, but that these files should go in a single directory; for instance, the declarations for `Acme::Trampoline` and `Acme::Skates::Rocket` would go in files named 'Acme::Trampoline.types' and 'Acme::Skates::Rocket.types'. A style of 'tree' specifies that each package should have its own trace file, and that those files should be arranged in directories whose structure matches the structure of their package names; in the example above, the files would be 'Acme/Trampoline.types' and 'Acme/Skates/Rocket.types'.

The default is 'tree'. Note that '--types-style' and '--decls-style' are currently constrained to be the same; if one is specified, the other will use the same value.

'--verbose'
'-v' Print additional information about what dfepl is doing, including external commands invoked.

7.6 Comma-separated-value front end convertcsv.pl

Daikon can process data from spreadsheets such as Excel. In order to use such files, first save them in comma-separated-value, also known as csv or comma-delimited or comma-separated-list, format. Then, convert the '.csv' file into a '.dtrace' file (and a '.decls' file) to be used by Daikon by running the `convertcsv.pl` program found in the '$DAIKONDIR/bin' directory. For example,

```
convertcsv.pl myfile.csv
```

produces files 'myfile.decls' and 'myfile.dtrace'.

Important: run `convertcsv.pl` without any arguments in order to see a usage message.

In order to ensure all data is processed, use Daikon with the '--nohierarchy' option, as follows:

```
java daikon.Daikon --nohierarchy myfile.decls myfile.dtrace
```

In a future release, the '--nohierarchy' option may not be necessary, but it should always be safe to use this option.

Before running `convertcsv.pl`, you may need to install Text::CSV, a Perl package that `convertcsv.pl` uses.

7.7 Other front ends

It is relatively easy to create a Daikon front end for another language or run-time system. For example, people have done this without any help at all from the Daikon developers. For more information about building a new front end, see section "New front ends" in *Daikon Developer Manual*.

A front end for the Eiffel programming language is distributed separately; see `http://se.inf.ethz.ch/people/polikarpova/citadel.html`.

A front end for the IOA programming language is distributed separately; see `http://groups.csail.mit.edu/tds/ioa.html`.

An earlier version of Daikon included a Lisp front end, but it is no longer supported.

An earlier version of Daikon provided a source-based front end for Java named dfej. It has been superseded by Chicory (see Section 7.1 [Chicory], page 88).

An earlier version of Daikon provided a source-based front end for C named dfec. It has been superseded by Kvasir (binary-based, for Linux/x86; see Section 7.3 [Kvasir], page 100) and Mangel-Wurzel (source-based, for all other platforms; see Section 7.4 [Mangel-Wurzel], page 122).

8 Tools for use with Daikon

This chapter describes various tools that are included with the Daikon distribution.

8.1 Tools for manipulating invariants

This section gives information about tools that manipulate invariants (in the form of '`.inv`' files).

8.1.1 Printing invariants

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

PrintInvariants is invoked as follows:

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

PrintInvariants shares many flags with Daikon. These flags are only briefly summarized here. For more information about these flags, see Section 4.4 [Daikon configuration options], page 20.

'`--help`' Print usage message.

'`--format` *`name`*'
: Produce output in the given format. See Section 5.1 [Invariant syntax], page 22.

'`--output_num_samples`'
: Output numbers of values and samples for invariants and program points; for debugging.

'`--ppt-select-pattern`'
: Only outputs program points that match the specified regular expression

'`--config` *`filename`*'
: Load the configuration settings specified in the given file. See Section 6.1 [Configuration options], page 52, for details.

'`--config_option` *`name=value`*'
: Specify a single configuration setting. See Section 6.1 [Configuration options], page 52, for details.

'`--dbg` *`category`*'
'`--debug`' Enable debug loggers.

'`--track` *`class<var1,var2,var3>@ppt`*'
: Track information on specified invariant class, variables and program point. For more information, also see section "Track logging" in *Daikon Developer Manual*.

8.1.2 MergeInvariants

The MergeInvariants utility merges multiple serialized invariant files to create a single serialized invariant file that contains the invariants that are true across each of the input files. The results of merging N serialized invariant files should be the same as running Daikon on the N original dtrace files.

MergeInvariants is invoked as follows:

```
java daikon.MergeInvariants [flags]... file1 file2...
```

file1 and *file2* are files containing serialized invariants produced by running Daikon. At least two invariant files must be specified.

MergeInvariants shares many flags with Daikon. These flags are only briefly summarized here. For more information about these flags, see Section 4.4 [Daikon configuration options], page 20.

'`-h --help`'
: Print usage message.

'`-o` *inv_file*'
: Output serialized invariants to the specified file; they can later be postprocessed, compared, etc. If not specified, the results are written to standard out.

'`--config_option` *name=value*'
: Specify a single configuration setting. See Section 6.1 [Configuration options], page 52, for details.

'`--dbg` *category*'
: Enable debug loggers.

'`--track` *class<var1,var2,var3>@ppt*'
: Track information on specified invariant class, variables and program point. For more information, also see section "Track logging" in *Daikon Developer Manual*.

8.1.3 Invariant Diff

The invariant diff utility is designed to output the differences between two sets of invariants. This is useful, for example, if you want to compare the invariants generated by two versions of the same program.

Invariant diff is invoked as follows:

```
java daikon.diff.Diff [flags]... file1 [file2]
```

file1 and *file2* are files containing serialized invariants produced by running Daikon or Diff with the '`-o`' flag. If *file2* is not specified, *file1* is compared with the empty set of invariants.

This section describes the optional flags.

'`--help`'
: Print usage message.

'`-d`'
: Display the tree of differing invariants (default). Invariants that are the same in *file1* and *file2* are not printed. At least one of the invariants must be justified. Does not print "uninteresting" invariants (currently some OneOf and Bound invariants).

'-u' Include "uninteresting" invariants in the tree of differing invariants.

'-y'
'--ignore_unjustified'
Include (statistically) unjustified invariants.

'-a' Display the tree of all invariants. Includes invariants that are the same in file1 and file2, and unjustified invariants.

'-s' For internal use only. Display the statistics between two sets of invariants. The pairs of invariants are placed in bins according to the type of the invariant and the type of the difference.

'-t' For internal use only. Display the same statistics as '-s', but as a tab-separated list.

'-m' Compute (*file1* - *file2*). This is all the invariants that appear in *file1* but not *file2*. Unjustified invariants are treated as if they don't exist. Output is written as a serialized InvMap to the file specified with the '-o' option. To view the contents of the serialized InvMap, run `java daikon.diff.Diff` *`file`*.

'-x' Compute (*file1* XOR *file2*). This is all the invariants that appear in one file but not the other. Unjustified invariants are treated as if they don't exist. Output is written as a serialized InvMap to the file specified with the '-o' option. To view the contents of the serialized InvMap, run `java daikon.diff.Diff` *`file`*.

'-n' Compute (*file1* UNION *file2*). This is all the invariants that appear in either file. If the same invariant appears in both files, the one with the better justification is chosen. Output is written as a serialized InvMap to the file specified with the '-o' option. To view the contents of the serialized InvMap, run `java daikon.diff.Diff` *`file`*.

'-o *inv_file*'
Used in combination with the '-m' or '-x' option. Writes the output as a serialized InvMap to the specified file.

'-j' For internal use only. Treat justification as a continuous value when gathering statistics. By default, justification is treated as a binary value — an invariant is either justified or it is not. For example, assume invariant I1 has a probability of .01, and I2 has a probability of .5. By default, this will be a difference of 1, since I1 is justified but I2 is not. With this option, this will be a difference of .49, the difference in the probabilities. This only applies when one invariant is justified, and the other is unjustified.

'-p' Examine all program points. By default, only procedure entries and combined procedure exits are examined. This option also causes conditional program points to be examined.

'-e' Print empty program points. By default, program points are not printed if they contain no differences.

'-v' Verbose output. Invariants are printed using the `repr()` method, instead of the `format()` method.

'-l' For debugging use only. Prints logging information describing the state of the program as it runs.

'--invSortComparator1 *classname*'
'--invSortComparator2 *classname*'
'--invPairComparator *classname*'
Use the specified class as a custom comparator. A custom comparator can be used for any of 3 operations: sorting the first set of invariants, sorting the second set of invariants, and combining the two sets into the pair tree. The specified class must implement the Comparator interface, and accept objects of type Invariant.

8.1.4 Annotate

The Annotate program inserts Daikon-generated invariants into Java source code as annotations in DBC, ESC, Java or JML format. These annotations are comments that can be automatically verified or otherwise manipulated by other tools. The Daikon website has an example of code after invariant insertion: `http://pag.csail.mit.edu/daikon/StackAr.html`.

Invoke Annotate like this:

```
java daikon.tools.jtb.Annotate Myprog.inv Myprog.java Myprog2.java ...
```

The first argument is a Daikon '`.inv`' or '`.inv.gz`' file produced by running Daikon with the '`-o`' command-line argument. All subsequent arguments are '`.java`' files. The original '`.java`' files are left unmodified, but Annotate produces new versions of the '`.java`' files (with names suffixed as '`-escannotated`', '`-jmlannotated`', or '`-dbcannotated`') that include the Daikon invariants as comments.

The options are:

'--format *name*'
Produce output in the given format. See Section 5.1 [Invariant syntax], page 22.

'--no_reflection'
Do not use reflection to find information about the classes being instrumented. This allows Annnotate to run without having access to the class files. Since the class files are necessary to generate "also" tags, those tags will be left out when this option is chosen.

'--max_invariants_pp *count*'
Output at most *count* invariants per program point (which ones are chosen is not specified).

'--wrap_xml'
Each invariant is printed using the given format (ESC, JML or DBC), but the invariant expression is wrapped inside XML tags, along with other information about the invariant.

For example, if this switch is set, the output format is ESC, and an invariant for method `foo(int x)` normally prints as

```
/* requires x != 0; */
```

Then the resulting output will look something like this (all in one line; we break it up here for clarity):

```
/* requires <INVINFO>
<INV> x != 0 </INV>
<SAMPLES> 100 </SAMPLES>
<DAIKON> x != 0 </DAIKON>
<DAIKONCLASS> daikon.inv.unary.scalar.NonZero </DAIKONCLASS>
<METHOD> foo() </METHOD>
</INVINFO> ; */
```

Note that the comment will no longer be a legal ESC/JML/DBC comment. To make it legal again, you must replace the XML tags with the string between the <INV> tag.

Also note the extra information printed with the invariant: the number of samples from which the invariant was inferred, the Daikon representation (i.e., the Daikon output format), the Java class that the invariant corresponds to, and the method that the invariant belongs to (`null` for object invariants).

If Annotate issues a warning message of the form

```
Warning: Annotate: Daikon knows nothing about field ...
```

then the Annotate tool found a variable in the source code that is was computed by Daikon. This can happen if Daikon was run omitting the variable, for instance due to –std-visibility. It can also happen due to a bug in Annotate or Daikon; if that is the case, please report it to the Daikon developers.

Known bug (logical shift in Java). Daikon's Java parser (adopted from javacc and JTB) accepts Java 1.5 syntax. An error in the new parser may produce illegal Java in the annotated file, if the source file to be annotated includes logical shift operators. See Section 9.6 [Known bugs], page 163.

8.1.5 AnnotateNullable

AnnotateNullable determines which variables in a Java program were ever null during execution. Its primary use is for performing inference for a type system that detects null dereference errors. An example is the Nullness checker that is part of the Checker Framework (`http://types.cs.washington.edu/checker-framework/`).

The Nullness checker requires a programmer to annotate some references with `@Nullable`, meaning the variable might be null; references that are left unannotated are never null at run time. (Alternately, the checker can use `@NonNull` for references that are never null, and leave unannotated for references that might be null. The Nullness checker supports either choice of default.) The checker then warns the programmer about possible null dereference errors.

The Nullness checker is useful, but writing all the annotations is tedious. The AnnotateNullable tool automatically and soundly determines a subset of the proper `@Nullable` annotations. Each annotation that it infers is correct. The programmer may need to write some additional `@Nullable` annotations, but that is much easier than writing them all.

To insert `@Nullable` annotations in your program, follow these steps:

1. Run your application one or more times to create a trace file. As always, the runs should exercise the application as thoroughly as possible. No incorrect `@Nullable` annotations are ever produced, but more thorough runs produce a larger number of `@Nullable` annotations.

```
java daikon.Chicory --dtrace-file=an.dtrace.gz your-command-and-options
```

2. Run Daikon on the resulting .dtrace file:

```
java daikon.Daikon an.dtrace.gz --no_text_output --config daikondir/java/daikon/an
```

   The example uses the 'annotate_nullable.config' configuration file that is included in the Daikon distribution. You may use any configuration file, or none, as long as the NonZero invariant is enabled. The 'annotate_nullable.config' enables *only* the NonZero invariant. This makes Daikon run much faster, but the resulting '.inv' file is useful only for the AnnotateNullable tool.

3. Run the AnnotateNullable tool to create an annotation index file. AnnotateNullable writes its output to standard out, so you should redirect its output to a '.jaif' file.

```
java daikon.AnnotateNullable an.inv.gz > nullable-annotations.jaif
```

4. Use the Annotation File Utilities (http://types.cs.washington.edu/annotation-file-utilities/) to insert the annotations in your .class or .java file.

```
insert-annotations mypackage.MyClass nullable-annotations.jaif
insert-annotations-to-source nullable-annotations.jaif mypackage/MyClass.java anno
```

AnnotateNullable is invoked as follows:

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

The flags are:

'-n --nonnull-annotations'
: Adds NonNull annotations as well as Nullable annotations. Unlike Nullable annotations, NonNull annotations are not guaranteed to be correct.

8.1.6 Runtime-check instrumenter (runtimechecker)

The runtimechecker instrumenter inserts, into a Java file, instrumentation code that checks invariants as the program executes. For a full list of options, run:

```
java daikon.tools.runtimechecker.Main help
```

The `instrument` command to runtimechecker creates a new directory 'instrumented-classes' containing a new version of the user-specified Java files, instrumented to check invariants at runtime and to record a list of invariant violations in a Java data structure.

Note that the instrumented program does not do anything with the list of violations; it merely creates the list. You will need to write your own code to process that list; see Section 8.1.6.1 [Accessing violations], page 148.

Here is an example of use of the runtime-check instrumenter. To create a version of file 'ubs/BoundedStack.java' that checks the invariants in invariant file 'BoundedStack.inv.gz', do:

```
java daikon.tools.runtimechecker.Main instrument BoundedStack.inv.gz \
    ubs/BoundedStack.java
```

The instrumented Java code references classes in the `daikon.tools.runtimechecker` package, so those classes must be present in the classpath when the instrumented classes are compiled and executed.

Invariants are evaluated at the program points at which they should hold. Three things can happen when evaluating an invariant:

- It evaluates to true, which means that the invariant holds. Program execution continues normally.
- It evaluates to false, which means that the invariant doesn't hold. In this case the corresponding `daikon.tools.runtimechecker.Property` is added to a list in the class `daikon.tools.runtimechecker.Runtime`. A programmer can obtain the growing list of violated invariants through the method `daikon.tools.runtimechecker.Runtime.getViolations()`. (See that class for other useful methods.)
- A `Throwable` (exception) is thrown when evaluating the invariant. In this case, the throwable is added to the list `daikon.tools.runtimechecker.Runtime.internalInvariantEvaluationE` The throwable is not rethrown.

8.1.6.1 Accessing violations

The instrumented class handles violations silently: it simply adds them to a list in the class `daikon.tools.runtimechecker.Runtime`. No "invariant violation" exceptions are thrown, and the violated invariants can only be obtained dynamically from class `daikon.tools.runtimechecker.Runtime`.

A future release of Daikon will provide tools that process the list in the following ways:

1. To write a file of all the violations for a program execution. A prototype of such a tool is provided in the Daikon distribution, as program `daikon.tools.runtimechecker.WriteViolationFile`. If you would ordinarily run your program as '`java MyProg arg1 arg2`', then running '`java daikon.tools.runtimechecker.WriteViolationFile MyProg arg1 arg2`' creates a file called '`violations.txt`' in the current directory. If the program under test calls `System.exit`, then no '`violations.txt`' file is created.
2. Throw an exception when any violation occurs.

The following code snippet contains a method `callMethod()` which presumably calls one of the methods in the instrumented class. The code detects if any violations occurred, and if so, prints a message.

```
daikon.tools.runtimechecker.Runtime.resetViolations();
daikon.tools.runtimechecker.Runtime.resetErrors();

callMethod();

List<Violation> vs = daikon.tools.runtimechecker.Runtime.getViolations();

if (!vs.isEmpty())
  System.out.println("Violations occurred.");
```

In addition, the instrumenter adds the following two methods to the instrumented class:

- `isDaikonInstrumented()`. Returns true (you could calling this method to see if the class has been instrumented).
- `getDaikonInvariants()`. Returns the array of properties being checked.

Known bug (logical shift in Java). Daikon's Java parser (adopted from javacc and JTB) accepts Java 1.5 syntax. An error in the new parser may produce illegal Java in the instrumented file, if the source file to be instrumented includes logical shift operators. See Section 9.6 [Known bugs], page 163.

8.1.7 InvariantChecker

The InvariantChecker program takes a set of invariants found by Daikon and a set of data trace files. It checks each sample in the data trace files against each of the invariants. Any sample that violates an invariant is noted, via a message printed to standard output or to a specified output file.

InvariantChecker is invoked as follows:

```
java daikon.tools.InvariantChecker [options] invariant-file dtrace-files
```

The *invariant-files* are invariant files ('`.inv`') created by running Daikon. The *dtrace-files* are data trace ('`.dtrace`') files created by running the instrumented program. The files may appear in any order; the file type is determined by whether the file name contains '`.dtrace`', or '`.inv`'.

The options are:

'`--help`' Print usage message.

'`--output` *output-file*'
Write any violations to the specified file.

'`--conf`' Checks only invariants that are above the default confidence level.

'`--filter`'
Checks only invariants that are not filtered by the default filters.

'`--dir` *directory-name*'
Processes all invariant files in the given directory and reports the number of invariants that failed on any of the dtrace files in that directory. We only process invariants above the default confidence level and invariants that have not been filtered out by the default filters.

'`--config_option` *name=value*'
'`--dbg` *category*'
'`--track` *class<var1,var2,var3>@ppt*'
These switches are the same as for Daikon. They are described in Chapter 4 [Running Daikon], page 17.

8.1.8 LogicalCompare

Given two sets of invariants describing the operation of a software module, or describing two implementations of a module with the same interface, we can define one set of invariants to be "stronger" than another roughly if in any situation where the "stronger" invariants hold, the "weaker" invariants also hold. The LogicalCompare tool examines two sets of invariants, and checks using the Simplify automatic theorem prover whether they satisfy a precise version of this relationship.

Simplify must be separately obtained (from `http://www.hpl.hp.com/downloads/crl/jtk/`) and installed in order to use this program.

The LogicalCompare program takes two mandatory arguments, which are '`.inv`' files containing invariants; the invariants will be checked to verify if the invariants in the first file are weaker (implied by) the invariants in the second file, and exceptions to this implication are printed. If no other regular arguments are supplied, all the method or function program points that exist in both files will be compared, with a exception message reported for each method that exists in the "weaker" set but not the "stronger". Alternatively, one or two additional arguments may be supplied, which name an `:::ENTER` program point and an `:::EXIT` program point to examine (if only an `:::ENTER` program point is supplied, the corresponding combined `:::EXIT` point is selected automatically). To be precise, for each pair of program points representing a single method or function, LogicalCompare will check that each precondition (`:::ENTER` point invariant) in the "stronger" invariant set is implied by some combination of invariants in the "weaker" invariant set, and that each postcondition (`:::EXIT` point invariant) in the "weaker" invariant set is implied by some combination of postconditions in the "stronger" set and preconditions in the "weaker" set. In summary, the syntax of an invocation of LogicalCompare will have the following form:

```
java daikon.tools.compare.LogicalCompare [options] \
       weak-invs strong-invs [enter-ppt [exit-ppt]]
```

LogicalCompare accepts the following options:

'`--assume` *file*'
: Read additional assumptions about the behavior of compared routines from the file *file*. The assumptions file should consist of lines starting with '`PPT_NAME`', followed by the complete name of an `:::ENTER` program point, followed by lines each consisting of a Simplify formula, optionally followed by a `#` and a human-readable annotation. Blank lines and lines beginning with a `#` are ignored. The assumption properties will be used as if they were invariants true at the strong `:::EXIT` point when checking weak `:::EXIT` point invariants.

'`--cfg` *option*`=`*value*'
: Specify a single configuration setting. The available settings are the same as can be passed to Daikon's '`--config_option`' option, though because the invariants have already been generated, some will have no effect. For a list of available options, see Section 6.1 [Configuration options], page 52.

'`--config-file=`*file*'
: Read configuration options from the file *file*. This file should have the same format as one passed to Daikon's '`--config`' option, though because the invariants have already been generated, some will have no effect.

'`--debug-all`'
'`--dbg` *category*'
: These options have the same effect as the '`--debug`' and '`--dbg`' options to Daikon, causing debugging logs to be printed.

'`--filters=[bBoOmjpi]`'
: Control which invariants are removed from consideration before implications are checked. Note that except as controlled by this option, LogicalCompare does not perform any of the filters that normally control whether invariants are printed by Daikon. Also, invariants that cannot be formatted for the Simplify automatic theorem prover will be discarded in any case, as there would be no other way to process them. Each letter controls a filter: an invariant is rejected

if it is rejected by any filter (or, equivalently, kept only if it passes through every filter).

'b' Discard upper-bound and lower-bound invariants (such as "x <= c" and "x >= c" for a constant c), when Daikon considers the constant to be uninteresting. Currently, Daikon has a configurable range of interesting constant: by default, -1, 0, 1, and 2 are interesting, and no other numbers are.

'B' Discard all bound invariants, whether or not the constants in them are considered interesting.

'o' Discard "one-of" invariants (which signify that a variable always had one of a small set of values at runtime), when the values that the variable took are considered uninteresting by Daikon.

'O' Discard all "one-of" invariants, whether or not the values involved are interesting.

'm' Discard invariants for which it was never the case that all the variables involved in the invariant were present at the same time.

'j' Discard invariants that Daikon determines to be statistically unjustified, according to its tests.

'p' Discard invariants that refer to the values of pass-by-value parameters in the postcondition, or to the values of objects pointed to by parameters in postconditions, when the pointer is not necessarily the same as at the entrance to the method or function. Usually such invariants reflect implementation details that would not be visible to the caller of a method.

'i' Discard implication invariants when they appear in `:::ENTER` program points.

The default set of filters corresponds to the letters '`ijmp`'.

'`--help`' Print a brief summary of available command-line options.

'`--no-post-after-pre-failure`'
If implication is not verified between two invariant sets after examining the preconditions, do not continue to check the implication involving postconditions. Because the postconditions aren't formally meaningful outside the domain specified by the preconditions, this is the safest behavior, but in practice trivial precondition mismatches may prevent an otherwise meaningful postcondition comparison. See also '`--post-after-pre-failure`'.

'`--proofs`'
For each implication among invariants that is verified, print a minimal set of conditions that establish the truth of the conclusion. The set is minimal, in the sense that if any condition were removed, the conclusion would no longer logically follow according to Simplify, but it is not the least such set: there may exist a smaller set of conditions that establish the conclusion, if that set is not a subset of the set printed. Beware that because this option uses a naive search technique, it may significantly slow down output.

'--post-after-pre-failure'
: Even if implication is not verified between two invariant sets after examining the preconditions, continue to check the implication involving postconditions. This is somewhat dangerous, in that if the implication does not hold between the preconditions, the invariant sets may be inconsistent, in which case reasoning about the postconditions is formally nonsensical, but the tool will attempt to ignore the contradiction and carry on in this case. This is now the default behavior, so the option has no effect, but it is retained for backward compatibility. See also '--no-post-after-pre-failure'.

'--show-count'
: Print a count of the number of invariants checked for implication.

'--show-formulas'
: For each invariant, show how it is represented as a logical formula passed to Simplify.

'--show-sets'
: Rather than testing implications among invariants, simply print the sets of weak and strong `:::ENTER` and `:::EXIT` point invariants that would normally be compared. The invariants are selected and filtered as implied by other options.

'--show-valid'
: Print invariants that are verified to be implied ("valid"), as well as those for which the implication could not be verified ("invalid" invariants, which are always printed).

'--timing'
: For each set of invariants checked, print the total time required for the check. This time includes both processing done by LogicalCompare directly, and time spent waiting for processing done by Simplify, but does not include time spent de-serializing the '.inv' input files.

8.2 DtraceDiff utility

DtraceDiff is a utility for comparing data trace ('.dtrace') files. It checks that the same program points are visited in the same order in both files, and that the number, names, types, and values of variables at each program point are the same. The differences are found using a content-based, rather than text-based, comparison of the two files.

DtraceDiff stops by signalling an error when it finds a difference between the two data trace files. (Once execution paths have diverged, continuing to emit record-by-record differences is likely to produce output that is far too voluminous to be useful.) It also signals an error when it detects incompatible program point declarations or when one file is shorter than the other.

DtraceDiff is invoked as follows:

```
java daikon.tools.DtraceDiff [flags] \
   [declsfiles1] dtracefile1  [declsfiles2] dtracefile2
```

Corresponding declarations ('.decls') files can optionally be specified on the command line before each of the two '.dtrace' files. Multiple '.decls' files can be specified. If no '`.decls`' file is specified, DtraceDiff assumes that the declarations are included in the '`.dtrace`' file instead.

DtraceDiff supports the following Daikon command-line flags:

'`--help`' Print usage message.

'`--config` *`filename`*'
: Load the configuration settings specified in the given file. See Section 6.1 [Configuration options], page 52, for details.

'`--config_option` *`name`*`=`*`value`*'
: Specify a single configuration setting. See Section 6.1 [Configuration options], page 52, for details.

'`--ppt-select-pattern=`*`ppt_regexp`*'
: Only process program points whose names match the regular expression.

'`--ppt-omit-pattern=`*`ppt_regexp`*'
: Do not process program points whose names match the regular expression. This takes priority over the '`--ppt-select-pattern`' argument.

'`--var-select-pattern=`*`ppt_regexp`*'
: Only process variables (whether in the trace file or derived) whose names match the regular expression.

'`--var-omit-pattern=`*`var_regexp`*'
: Ignore variables (whether in the trace file or derived) whose names match the regular expression, which uses Perl syntax. This takes priority over the '`--var-select-pattern`' argument.

DtraceDiff uses appropriate comparisons for the type of the variables in each program point being compared. In particular:

- Hashcode (pointer or address) values may differ from one run of the same program to the next, and there may not be a one-to-one mapping of hashcode values between different program executions, so the comparison function only looks for null versus non-null pointer values.
- Floating-point values are subject to roundoff error from printing and reading, so they are compared with a "fuzzy" rather than exact equality test.

9 Troubleshooting

This chapter gives solutions for certain problems you might have with Daikon. It also tells you how to report bugs in a useful manner.

If, after reading this section and other parts of the manual, you are unable to solve your problem, you may wish to send mail to one of the mailing lists (see Section 1.1 [Mailing lists], page 1).

9.1 Problems running Daikon

You may find the debugging flags '`--debug`' and '`--dbg` *category*' useful if you wish to track down bugs or better understand Daikon's operation; See Section 4.5 [Daikon debugging options], page 21. See Section 6.1 [Configuration options], page 52, for another way to adjust Daikon's output.

9.1.1 Too much output

Sometimes, Daikon may produce a very large number of seemingly irrelevant properties that obscure the facts that you were hoping to see. Which properties are irrelevant depends on your current task, so Daikon provides ways for you to customize its output. See Daikon's command-line options (see Chapter 4 [Running Daikon], page 17), and the techniques for enhancing its output (see Chapter 6 [Enhancing Daikon output], page 52), including its configuration options (see Section 6.1 [Configuration options], page 52). The options for the front ends — such as Chicory (see Section 7.1.1 [Chicory options], page 88) and Kvasir (see Section 7.3.2 [Kvasir options], page 101) — give additional control.

Some irrelevant properties are over unrelated variables, like comparing an array index to elements of the array. You should always use the DynComp tool (see Section 7.2 [DynComp for Java], page 94, Section 7.3.3 [DynComp for C/C++], page 107) to avoid producing such properties.

Some irrelevant properties are not relevant to the domain (e.g., bitwise operations). You can exclude whole classes of unhelpful invariants from Daikon's output (see Section 6.1.1.2 [Options to enable/disable specific invariants], page 53).

Some irrelevant properties are over variables you do not care about, or are in parts of the program that you do not care about. You can exclude certain variables or procedures from Daikon's output (see Section 4.3 [Processing only part of the trace file], page 20 and Section 6.1.1.4 [Options to enable/disable derived variables], page 69).

Some irrelevant properties are logically redundant — multiple properties express the same facts in different ways. You can eliminate such properties from Daikon's output (see Section 4.2 [Options to control invariant detection], page 19).

Some irrelevant output indicates a deficiency in your test suite: your test suite is so small that many arbitrary properties hold over it. This happens when the test suite does not execute the code with a broad distribution of values, but only executes the code with a few specific values. This problem disappears if you augment your test suite so that it exercises the code with more different values.

More generally, each property that Daikon produces is a true fact about how the target program behaved. However, some of these facts would be true for any execution of the target program, and others are accidents of the particular executions that Daikon observed. Both types of facts may be useful: the former tell you about your program, and the latter tell you about your test suite (and how to improve it!).

9.1.2 Missing output invariants

Daikon will sometimes not output invariants that are expected. There are a number of reasons why this may happen:

- There is a sample that violate the invariant
- The invariant is true, but does not pass one of the output filters
- One or more of the variables in the invariant always has the same value as another variable. Daikon only prints invariants over one variable (the leader) from the set of equal variables (see Section 5.4.2 [Equal variables], page 27).
- The program point had no samples (see Section 9.1.3 [No samples], page 155).

There are two command line options (–disc_reason and –track) that will display information about invariants that are not printed. The *–disc_reason* option will indicate why a particular invariant was discarded in most cases. If it does not provide enough information, try the *–track* option which traces the invariant through all of Daikon's processing steps. See Section 4.5 [Daikon debugging options], page 21 for more information.

Note that in each case the description (class, variables, program point) of the invariant must be entered carefully. It may be helpful to try the option on a similar invariant that is printed to make sure that each is specified correctly.

9.1.3 No samples and no output

When Daikon produces no output, that is usually a result of it having no samples from which to generalize. Use the '`--output_num_samples`' flag to Daikon to find out how many samples it is observing. This section tells you how to debug your problem if the answer is 0, but you believe that there are samples in the file you are feeding to Daikon.

Using the normal dataflow hierarchy, Daikon only explicitly processes `:::EXIT` program points. Other program points, such as `:::ENTER` program points, are processed indirectly when their corresponding `:::EXIT` points are encountered. (You can disable this behavior with the '`--nohierarchy`' switch to Daikon; see See Section 4.2 [Options to control invariant detection], page 19.) If no `:::EXIT` program points are present (perhaps every execution threw an exception, you filtered out all the `:::EXIT` program points, or the data trace is obtained from spreadsheet data instead of from a program execution), then Daikon will not process any of the other program points, such as the `:::ENTER` program points. You can make Daikon print information about unmatched procedure entries via the '`daikon.FileIO.unmatched_procedure_entries_quiet`' configuration option (see Section 6.1.1.6 [General configuration options], page 73).

Another way to increase the number of invariants printed out is to lower the confidence bound cutoff. Daikon only prints invariants whose confidence level is greater than the bound specified by the '`--conf_limit`' option (see Section 4.2 [Options to control invariant detection], page 19). In order to maximize the number of invariants printed, use '`--conf_limit 0`' to see all invariants Daikon is considering printing.

To try to determine why an invariant is not printed, use the '`--track`' to determine why Daikon does not print an invariant (see Section 4.5 [Daikon debugging options], page 21).

9.1.4 No return from procedure

Daikon sometimes issues a warning that a procedure in the target program was entered but never exited (that is, the target program abnormally terminated). In other words, the '.dtrace' file contains more entry records than exit records for the given procedure. Some procedures that were entered were never recorded to have exited: either they threw an exception, skipping the instrumentation code that would have recorded normal termination, or the target program's run was interrupted. When this happens, the entry sample is ignored; the rationale is that the particular values seen led to exception exit, were probably illegal, and so should not be factored into the method preconditions.

In some cases, exceptional exit from a procedure can cause procedure entries and exits (in the trace file) to be incorrectly matched up; if they are incorrectly matched, then the `orig(x)` values may be incorrect. Daikon has two techniques for associate procedure exits with entries — the nonce technique and the stack technique. If a '.dtrace' file uses the nonce technique, `orig(x)` values are guaranteed to be correct. If a '.dtrace' file uses the stack technique, then incorrect `orig(x)` values are likely to occur. You can tell which technique Daikon will use by examining the '.dtrace' file. If the second line of each entry in the '.dtrace' file is 'this_invocation_nonce', then Daikon uses the nonce technique. Otherwise, it uses the stack technique. Which technique is used is determined by the front end, which creates the '.dtrace' file, and typically cannot be controlled by the user.

9.1.5 Unsupported class version

Daikon requires a Java 5 JVM (see Section 2.2.1 [Requirements], page 2). An error such as

```
Exception in thread "main" java.lang.UnsupportedClassVersionError:
daikon/Daikon (Unsupported major.minor version 49.0)
```

indicates that you are trying to run Daikon on an older JVM. You need to install a newer version of Java in order to run Daikon.

9.1.6 Out of memory

If Daikon runs out of memory, generating a message like

```
Exception in thread "main" java.lang.OutOfMemoryError
        <<no stack trace available>>
```

then it is likely that it has created more invariants than will fit in memory. The number of invariants created depends on the number of program points and the number of variables at each program point. In addition to the solutions discussed in Section 9.2.1 [Reducing program points], page 160, you can try increasing the amount of memory available to Java with the '-mx' argument to `java`. (This flag is JVM-specific; see your JVM documentation for details. For instance, its correct name in JDK versions 1.3 and later is '-Xmx'.) However, the value you use should be less your system's total amount of physical memory. The default may be 64 megabytes or less; to permit use of up to 256 megabytes, you would run Java like so:

```
java -mx256m ...
```

When using the Java HotSpot JVM, an additional parameter may need to be increased. HotSpot uses a separately-limited memory region, called the "permanent generation", for several special kinds of allocation, one of which (interned strings) Daikon sometimes uses heavily. It may be necessary to increase this limit as well, with the '-XX:MaxPermSize='

option. For instance, to use 512 megabytes, of which at most 256 can be used for the permanent generation, you would run Java like so:

```
java -Xmx512m -XX:MaxPermSize=256m
```

Another possible problem is the creation of too many derived variables. If you supply the '`--output_num_samples`' option to Daikon (see Section 4.1 [Options to control Daikon output], page 17), then it will list all variables at each program point. If some of these are of no interest, you may wish to suppress their creation. For information on how to do that, see Section 6.1.1.4 [Options to enable/disable derived variables], page 69. Also see Section 9.2.2 [Reducing variables], page 160 for other techniques.

Any output generated before the out-of-memory error is perfectly valid.

9.1.7 Simplify errors

The warning "Could not utilize Simplify" indicates that the Simplify theorem-prover could not be run; this usually indicates that the Simplify binary was not found on the user's path. Simplify must be separately obtained (from `http://www.hpl.hp.com/downloads/crl/jtk/`) and installed.

If Simplify is not used, certain redundant (logically implied) invariants may appear in Daikon's output. The output is correct, but more verbose than it would be if you used Simplify.

9.1.8 Contradictory invariants

The invariants Daikon produces are all true statements about the supplied program executions, so they should be mutually consistent. Sometimes, however, because of a bug or a limitation in Daikon, contradictory invariants are produced.

One known problem involves object invariants. Daikon infers object invariants by observing the state of an object when its public methods are called. However, if an object has publicly accessible fields that are changed by code outside the class, after which no public methods are called, invariants about the state of the object as seen by other code can contradict the class's object invariants. A workaround is to allow changes to an object's state from outside the class only by way of public methods.

Besides confusing the user, contradictory invariants also cause trouble for the Simplify theorem prover that implements the '`--suppress_redundant`' option. When the invariants at a particular program point contradict each other or background information (such as the types of objects), Simplify becomes unable to distinguish redundant invariants from non-redundant ones.

The best solution in such cases is to fix the underlying cause of the contradictory invariants, but since that is sometimes not possible, Daikon will try to work around the problem by avoiding the invariants that cause a contradiction. Daikon will attempt to find a small subset of the invariants that aren't mutually consistent, and remove one, repeating this process until the remaining invariants are consistent. (Note that the invariants are removed only for the purposes of processing by Simplify; this does not affect whether they will be printed in the final output). While this technique can allow redundant invariants to be found when they otherwise wouldn't be, it has some drawbacks: the choice of which invariant to remove is somewhat arbitrary, and the process of finding contradictory subsets can be time consuming. The removal process can be disabled with the '`daikon.simplify.LemmaStack.remove_contradictions`' configuration option.

9.1.9 Method needs to be implemented

Daikon may produce output like the following (but all on one line):

```
method daikon.inv.binary.twoSequence.SubSequence.format_esc()
needs to be implemented:
this.theArray[0..this.topOfStack] is a subsequence of
orig(this.theArray[0..this.topOfStack])
```

This indicates that a particular invariant (shown on the last two lines above) cannot be formatted using the current formatting. In this example, the invariant can be formatted using Daikon's default formatting (which is how it is shown above), but (as of April 2002) Daikon cannot output it in ESC format, so Daikon prints the above message instead. The message also shows exactly what Java method needs to be implemented to correct the problem. You can ignore such messages, or else use an output formatting that can handle those invariants. Annotate (see Section 8.1.4 [Annotate], page 145) automatically ignores unformattable invariants.

9.1.10 Daikon runs slowly

Daikon's runtime and space depend on the particular data that it analyzes. Informally, invariant detection time can be characterized as

```
O((vars^3 * falsetime + trueinvs * testsuite) * procedures)
```

where *vars* is the number of variables *at a program point*, *falsetime* is the (small constant) time to falsify a potential invariant, *trueinvs* is the (small) number of true invariants at a program point, *testsuite* is the size of the test suite, and *procedures* is the number of instrumented program points. The first two products multiply a number of invariants by the time to test each invariant.

If there are many true invariants over an input, then Daikon continues to check them all over the entire input. By contrast, if not many invariants are true, then Daikon need no longer check them once they are falsified (which in practice happens quickly). Daikon processes each procedure independently.

Another important factor affecting Daikon's runtime is the number of variables. Because invariants involve up to three variables each, the number of invariants to check is cubic in the number of variables at a single program point. Derived variables (such as `a[i]`, introduced whenever there is both an array `a` and an integer `i`) can increase the number of variables substantially. The '`daikon.derive.Derivation.disable_derived_variables`' and individual '`daikon.derive.*.*.enabled`' configuration variables (see Section 6.1.1.4 [Options to enable/disable derived variables], page 69) may be used to disable derived variables altogether or selectively, at the cost of detecting fewer invariants, especially over sequences.

For details on improving Daikon's performance, see Section 9.1.6 [Out of memory], page 156, and Section 9.2 [Large dtrace files], page 159.

9.1.11 Bigger traces cause invariants to appear

Suppose that you run Daikon twice. The first time, you supply Daikon with traces T. The second time you supply Daikon with traces T+T': either more files, or file(s) that are supersets of the original one(s). The second Daikon execution may report fewer invariants, more invariants, or a mix.

The second execution may report **fewer** invariants, because the additional data has eliminated overfitting (false positives). There may have been some accidental property of the shorter executions that is not true in the longer ones.

Even though fewer invariants are true on the second execution, Daikon may report invariants that it did not report on the first execution. We mention two reasons that Daikon might not report an invariant that is true: statistical justification, and implication.

The second execution may report **more** invariants, because of Daikon's statistical justification tests. Daikon only reports a property if it is statistically justified, and Daikon needs to see enough samples for the statistical test to work. So, there may have been a property that was true both in the short trace and in the long one, but Daikon only reported it for the long one. If you want to prevent this from happening, you can adjust the confidence limit so that the property is reported even in the short executions; see the command-line option `--conf_limit`. For instance, supplying `--conf_limit 0` causes all properties that have not been falsified to be printed.

Another reason that the second execution might report **more** invariants (more specifically, might report an invariant that did not appear in the first execution) is because Daikon does not report redundant, or implied, invariants. Suppose that both `i < j` and `i <= j` were true on the first execution. Daikon would report only `i < j`; Daikon would not report `i <= j`, which is implied by what Daikon has reported. Further suppose that the second execution had a sample containing `i=22, j=22`. Only `i <= j` would be true in the second execution, and Daikon would report it. (The invariant `i < j` is an example of a false positive or overfitting in the first execution.)

9.2 Large data trace (.dtrace) files

Running instrumented code can create very large '`.dtrace`' files. This can be a problem because writing the large files can slow the target programs substantially, or because the large files may fill up your disk. The solution is to create smaller data trace files, by computing invariants over fewer program points (procedures), computing invariants over fewer variables, or computing invariants over fewer samples (executions).

It is usually possible to create an '`.inv`' file equivalent to the one that Daikon would have computed, had Daikon been able to process your entire program over its full test suite. First, use the techniques below (see Section 9.2.1 [Reducing program points], page 160) to split your '`.dtrace`' file into parts. Next, run Daikon on each resulting '`.dtrace`' file. Finally, use the Section 8.1.2 [MergeInvariants], page 143 tool to combine the resulting '`.inv`' files into one.

Independently from the above techniques, you can run Daikon online. The term "online execution" refers to running Daikon at the same time as the target program, without writing any information to a file. This can avoid some I/O overhead, it prevents filling up your disk with files, and in the future Daikon may be able to produce partial results as the target program is executing. The Kvasir front end supports online execution via use of (normal or named) Unix pipes (see Section 7.3.7 [Online execution], page 119).

9.2.1 Reducing program points (functions)

Here are ways to compute invariants over a subset of the program points (functions) in your program.

1. You can make Daikon ignore some program points. With the '`--ppt-select-pattern=`*`ppt_regexp`*' flag (see Section 4.3 [Processing only part of the trace file], page 20), only program points matching the regular expression are processed. Likewise, the '`--ppt-omit-pattern=`*`ppt_omit_regexp`*' option causes program points matching the regular expression to be ignored.

   Also, the configuration variable '`daikon.Daikon.ppt_perc`' allows a percentage of the program points to be processed. See Section 6.1.1.6 [General configuration options], page 73, for details.
2. You can remove some program points (functions) from your '`.dtrace`' file. The `trace-purge-fns.pl` script takes as arguments a (Perl) regular expression and a list of files. It modifies each file in place, removing every program point (function) whose name matches the regular expression. The '`-v`' flag means to retain rather than discard matching program points. For instance, to create two subparts of a '`.dtrace`' file — one containing the getters and setters, and the other containing all other functions — use the following commands:

   ```
   cp myprog.dtrace myprog-setters.dtrace
   trace-purge-fns.pl -v 'set|get' myprog-setters.dtrace
   cp myprog.dtrace myprog-non-setters.dtrace
   trace-purge-fns.pl 'set|get' myprog-non-setters.dtrace
   ```
3. You can instrument fewer methods, creating smaller '`.dtrace`' files in the first place (rather than cutting the '`.dtrace`' files down afterward).
   - With Chicory, use the '`--ppt-omit-pattern`' or '`--ppt-select-pattern`' options (see Section 4.3 [Processing only part of the trace file], page 20, Section 7.1.1 [Chicory options], page 88) to restrict which program points are traced. Running the instrumented program will result in a smaller '`.dtrace`' file that contains fewer records.
   - With Kvasir, use the '`--ppt-list-file`' option to specify a list of program points that should be traced (see Section 7.3.4 [Tracing only part of a program], page 109 section for more details).
4. If you are using a source-based front end, you can instrument fewer files.

9.2.2 Reducing variables

Here are ways to compute invariants over a subset of the variables in your program. This changes the resulting invariants, because invariants over the missing variables (including any relationship between a missing variable and a retained variable) are not detected or reported. For instance, you might remove uninteresting variables (or ones that shouldn't be compared to certain others) or variables that use a lot of memory (such as some arrays).

1. You can make Daikon ignore certain variables rather than modifying the '`.dtrace`' file directly. Analogously with the '`--ppt-select-pattern`' and '`--ppt-omit-pattern`'

flags, the '`--var-select-pattern`' and '`--var-omit-pattern`' flags restrict which variables Daikon processes (see Section 4.3 [Processing only part of the trace file], page 20, and Section 7.1.1 [Chicory options], page 88).

2. You can reduce the number of variables that are output by instrumented code — for instance, output '`a`' and '`a.b`' but not '`a.b.c`'. Do this by reducing the class/structure instrumentation depth.
   - With Chicory, use the '`--nesting-depth=N`' option.
   - With Kvasir, use the '`--struct-depth=N`' or the '`--nesting-depth=N`' option.
   - With Mangel-Wurzel, use the '`--struct-depth=N`' or the '`--nesting-depth=N`' option.
3. With Kvasir, you can either ignore all global and/or static variables with the '`--ignore-globals`' and '`--ignore-static-vars`' options or manually specify a subset of variables to trace using the '`--var-list-file`' option (see Section 7.3.4 [Tracing only part of a program], page 109 section for details)
4. You can pare down an existing '`.dtrace`' file using the `trace-purge-vars.pl` script. Analogously to the `trace-purge-fns.pl` script, it removes certain variables from all program points in a function (or retains them, with the '`-v`' flag). After running this command, you will need to edit the corresponding '`.decls`' file by hand to remove the same variables.

9.2.3 Reducing executions

Here are ways to run Daikon over fewer executions of each program point. (You cannot combine the resulting invariants in order to obtain the same result as running Daikon over all the executions.)

1. If you have multiple '`.dtrace`' files (perhaps resulting from multiple program runs), you can run Daikon on just some of them.
2. You can terminate the instrumented program when it has created a sufficiently large '`.dtrace`' file. If you interrupt the program while it is in the middle of writing a record to the '`.dtrace`' file, the last record may be only partially written. Use the `daikon/bin/trace-untruncate` program to remove the last, possibly partial, record from the file:

   ```
   trace-untruncate myfile.dtrace
   ```

   modifies '`myfile.dtrace`' in place to remove the last record.

   Alternately, you can use the `daikon/bin/trace-untruncate-fast` program. It operates much faster on very large files. In order to use `trace-untruncate-fast`, you must have already compiled it (see Chapter 2 [Installing Daikon], page 2).

9.3 Parsing Java 5.0 code

The Annotate (see Section 8.1.4 [Annotate], page 145) and runtimechecker (see Section 8.1.6 [Runtime-check instrumenter], page 147) tools use an external library (JTB, `http://compilers.cs.ucla.edu/jtb/`) for parsing Java code. That external library has an error in parsing logical shift operators such as `x << y` or `x >> y`. Therefore, such invariants (along with a few other invariants that cannot be properly formatted as Java code) are suppressed from the output of these tools. (Annotate inserts them as comments, and runtimechecker silently ignores them.)

9.4 Problems with Chicory

Before reporting or investigating a problem with Chicory, always check that the program executes properly when not being run under Chicory's control.

For example, if a command such as

```
java daikon.Chicory DataStructures.StackArTester
```

fails with an error, then first try

```
java DataStructures.StackArTester
```

which is likely to fail with the same error.

If the latter command also fails, the problem is not with Chicory. First solve your Java problem, then once again attempt to use Chicory.

If the latter command does not fail, then you have found a bug in Chicory; please report it if it is not already explained in this manual.

9.4.1 VerifyError constant pool index error

If Chicory throws an error such as the following:

```
Exception in thread "main" java.lang.VerifyError:
  (class: ps1/PublicTest, method: <init> signature: (Ljava/lang/String;)V)
  Illegal constant pool index
```

then the problem is most likely that your classpath contains a version of the BCEL library ('bcel.jar') that is not compatible with Java 5. You should either remove that version of BCEL from your classpath, or you should ensure that it appears after 'daikon.jar'. (If you are running Daikon from sources rather than from 'daikon.jar', then ensure that '$DAIKONDIR/java/lib/bcel.jar' is the first version of BCEL on your classpath.)

9.5 Reporting problems

If you have any questions, can suggest ways to improve the documentation, find bugs in the system, or have suggestions for its improvement, please send email to daikon-developers@lists.csail.mit.edu. (If you can't figure out how to do something or do not understand why Daikon works the way it does, that is a bug, too — in the Daikon documentation. Please report those as well.) We will try to assist you and to correct any problems, so please don't hesitate to ask for help or report difficulties. Additionally, if you can contribute enhancements or bug fixes, those will be gratefully accepted.

In order for us to assist you, please provide a complete and useful bug report. Your bug report must provide all the information that is required in order to replicate the bug and verify that our fix corrects the problem. If you do not provide complete information, we will not be able to assist you.

Your bug report should include:

- the version of Daikon, which appears in the file 'daikon/README' and is also printed when you run Daikon. If you are not using the most recent version, download a newer version from http://pag.csail.mit.edu/daikon/ to see whether your problem has already been corrected. If you are using a modified version of Daikon, you should verify that the problem exists in Daikon as distributed.

- a description of exactly what you did (in sufficient detail for others to reproduce the problem), exactly what happened, and what you expected to happen instead. One good way to describe what you did is a list of commands that, if executed, reproduces your error. A good way to show what happened is a transcript of execution of all of the commands. (A list of commands and a transcript are **much** more useful than a vague description; please don't give vague English when you can supply a more precise specification instead.) It is crucial that you not omit steps in your report. For example, include instructions for installing your software and all customizations to the software or your environment, including all relevant environment variables. Please do not force the developers to speculate about what you did; that is a waste of their time, since you already have the knowledge.
- input files that permit the problem to be replicated (by following the detailed steps in your bug report). The most important thing is the original, uninstrumented source files (e.g., '`.java`'), and any inputs/tests used when you ran the program. It is also helpful to include instrumented source files, '`.decl`' files, and '`.dtrace`' files.
- the operating system and revision you are using (e.g., Debian stable, Windows XP service pack 2, etc)
- any other information that you consider relevant.

When users provide an inadequate bug report, it is frequently more difficult for us to reproduce an error than to correct it. If you make it easy for us to reproduce and verify the problem, then it is much more likely to be corrected. Thanks for helping us to help you!

You may also wish to take advantage of the Daikon mailing lists (see Section 1.1 [Mailing lists], page 1).

9.6 Known bugs

- This problem affects Section 8.1.4 [Annotate], page 145 and the Section 8.1.6 [Runtime-check instrumenter], page 147, which output Java source code. If the java file to be annotated/instrumented uses logical shift operators (`>>`, `<<`, and `>>>`), the annotated/instrumented java may contain extra brackets in these operators. For example, the expression `x << y` in the input source file may erroneously be translated as `x <<<<<< y` in the annotated/instrumented file.

  This error's presence will be immediately apparent if you try to compile the annotated/instrumented file and it fails to compile because the extra brackets make it syntactically illegal. In this case, the best solution we can recommend is that you manually fix the occurrence of the logical shift operator.

  This error is caused by a bug in the parser generator that generates Daikon's Java parser. We expect it will be fixed by the next release.

9.7 Further reading

More information on Daikon can be found in the Daikon Developer Manual (see section "Top" in *Daikon Developer Manual*). For instance, the Daikon Developer Manual indicates how to extend Daikon with new invariants, new derived variables, and front ends for new languages. It also contains information about the implementation and about how to debug.

You may find discussions on the mailing lists (see Section 1.1 [Mailing lists], page 1) helpful. The mailing list archives may contain helpful information, but we strive to incorporate that information in this manual so that you don't have to search the archives as well.

For further reading, see the list of publications at the Daikon homepage, `http://pag.csail.mit.edu/daikon/pubs/`.

10 Details

The Daikon invariant detector is named after an Asian radish. “Daikon” is pronounced like the combination of the two one-syllable English words “die-con”.

More information on Daikon can be found in the Daikon Developer Manual (see section “Top” in *Daikon Developer Manual*). For instance, the Daikon Developer Manual indicates how to extend Daikon with new invariants, new derived variables, and front ends for new languages. It also contains information about the implementation and about debugging flags.

10.1 History

This manual describes Daikon version 4.6.3, released December 18, 2009. A more detailed list of revisions since mid-2001 can be found in file ‘`daikon/doc/CHANGES`’ in the distribution; this section gives a high-level view of the package’s history.

There have been four major releases of Daikon, with different features and capabilities. User experiences and technical papers should be judged based on the version of Daikon current at the time of use.

Daikon 1 was written in the Python programming language in 1998. It included front ends for C, Java, and Lisp. The C front end was extremely limited and failed to operate correctly on all C programs: sometimes it suffered a segmentation fault while instrumenting a target program, and even when that did not happen, sometimes the instrumented program segmentation-faulted while running. The Lisp front end operated correctly on all Lisp programs, but only instrumented certain common constructs, leaving other language features uninstrumented. The Java front end was reasonably reliable. The Lisp front end instrumented procedure entries, exits, and loop heads; the C front ends instrumented only procedure entries and exits; and the Java front end instrumented program points for object invariants as well as procedure entries and exits.

Daikon 2 was a complete rewrite in the Java programming language and was the first version to contain a substantive manual. Daikon 2 uses the same source-based Java front end as did Daikon 1, though with certain enhancements. Its C front end was rewritten from scratch; it instruments only procedure entries and exits. A front end also exists for the IOA programming language, but is not included in the Daikon distribution.

Daikon 3 is a redesign of the invariant detection engine to work incrementally — that is, to examine each sample (execution of a program point) once, then discard it. By contrast, Daikon 1 and Daikon 2 made multiple passes over the data. This simplified their algorithms but required storing all the data in memory at once, which was prohibitive, particularly since data trace files may be gigabytes in size. Daikon 3 also introduces the idea of a *dataflow hierarchy*, a way to relate and connect program points based on their variables.

Daikon 4 includes new binary front ends for Java and for C. These front ends make Daikon much easier to use. Daikon 4 makes ‘`.decls`’ files optional; program point declarations are permitted to appear in ‘`.dtrace`’ files. Daikon 4 is released under more liberal licensing conditions that place no restrictions on use.

10.2 License

Daikon uses the Java port of the GNU getopt library, which is copyright 1998 Aaron M. Renn. The getopt library is free software, and may be redistributed or modified under the terms of the GNU Library General Public License version 2. A copy of this license is included with the Daikon distribution as the file '`java/gnu/getopt/COPYING.LIB`'.

Daikon's unit tests use the JUnit testing framework, which is governed by the Common Public License, version 1.0. JUnit is provided on an "as is" basis, without warranties or conditions of any kind, either express or implied including, without limitation, any warranties or conditions of title, non-infringement, merchantability or fitness for a particular purpose. Neither the Daikon developers nor the authors of the JUnit framework shall have any liability for any direct, indirect, incidental, special, exemplary, or consequential damages (including without limitation lost profits), however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use or distribution of JUnit or the exercise of any rights granted in the Common Public License, even if advised of the possibility of such damages. Those portions of JUnit that appear in the Daikon distribution may be redistributed under the same terms as Daikon itself; this offer is made by the Daikon developers exclusively and not by any other party. The Common Public License is included with the Daikon distribution as the file '`java/junit/cpl-v10.html`'.

Note that the front ends discussed in this manual are separate programs, and some are made available under different licenses. Because the front ends are separate programs not derived from the Daikon invariant detection tool, you are neither required nor entitled to use the Daikon invariant detector itself under these other licenses.

10.3 Mailing lists reminder

If you use Daikon, please subscribe to the daikon-announce and daikon-discuss mailing lists (see Section 1.1 [Mailing lists], page 1). The daikon-announce list will inform you of new versions, enhancements, and bug fixes. On the daikon-discuss mailing list, you can obtain help from, and offer help to, other users. We would also appreciate a brief description of how you are using Daikon, sent to `daikon-developers@lists.csail.mit.edu`. We are curious about how users exploit Daikon, and we are eager for anecdotes about its successes and failures, so that we can make Daikon more effective for its users.

10.4 Credits

The following individuals have contributed to Daikon: Yuriy Brun, Jake Cockrell, David Cok, Adam Czeisler, Brian Demsky, Alan Donovan, Nii Dodoo, Alan Dunn, Michael Ernst, Eric Fellheimer, William Griswold, Philip Guo, Melissa Hao, Michael Harder, Dieter von Holten, Greg Jay, Josh Kataoka, Lee Lin, Sandra Loosemore, Vikash Mansinghka, Stephen McCamant, Samir Meghani, Benjamin Morse, Jelani Nelson, Ryan Newton, Jeremy Nimmer, Toh Ne Win, David Notkin, Carlos Pacheco, Jeff Perkins, Jaime Quinonez, Robert Rudd, Alexandru Salcianu, Kathryn Shih, Matthew Tschantz, Iuliu Vasilescu, Chen Xiao, Tao Xie, Jeff Yuan.

Craig Kaplan carved the Daikon logo.

The feedback of Daikon users has been very valuable. We are particularly grateful to B. Thomas Adler, Rich Angros, Tadashi Araragi, Seung Mo Cho, Christoph Csallner, Dorothy Curtis, Diego Garbervetsky, Mangala Gowri, Madeline Hardojo, Engelbert Hubbers, Nadya Kuzmina, Scott McMaster, Charles O'Donnell, Alex Orso, Rodric Rabbah, Manos Renieris, Rosie Wacha. Many others have also been generous with their feedback, for which we are also grateful.

If your name has been inadvertently omitted from this section, please let us know so we can correct the oversight.

Financial support has been provided by: National Science Foundation (NSF), Defense Advanced Research Projects Agency (DARPA), ABB, Edison Design Group, IBM, NTT, MIT Oxygen Project, Raytheon, Toshiba.

10.5 Citing Daikon

If you wish to cite Daikon in a publication, we recommend that you reference one of the scholarly papers listed at `http://pag.csail.mit.edu/daikon/pubs/#invariant-detection` in lieu of, or in addition to, referencing this manual and the Daikon website (`http://pag.csail.mit.edu/daikon/`).

Index

-

.

/

:

@

[

6

A

B

C

D

N

O

P

R

S

T

U

V

W

X