DIDUCE, v 1.0Beta

User's Manual

http://diduce.sf.net

Questions or Comments ?
Please send email to diduce-users@lists.sf.net
or
hangal@cs.stanford.edu

Table of Contents

1.Overview

DIDUCE (Dynamic Invariants Detection ∪ Checking Engine) is a tool to help detect bugs in Java programs. DIDUCE works by observing the behaviour of a program on test cases for which it works correctly; this behaviour helps formulate a theory about the invariants that the program obeys. DIDUCE then checks these invariants on another testcase for which a program fails. This model of debugging is most useful for debugging regression testcase failures, i.e. when you have a program which runs correctly on some or most inputs, but crashes on other inputs which hit corner cases. It automates a lot of debugging by isolating "what's different" about this test case compared to the ones that passed.

DIDUCE is especially useful in catching complex corner case bugs which occur after the program has been running for a while. DIDUCE is also good at pinpointing root causes of bugs in situations which the programmer or tester is unfamiliar with large parts of the code. Users have successfully used DIDUCE to quickly and effortlessly detect bugs in programs of which they had no initial understanding.

DIDUCE has also detected bugs in situations where it was not apparent through the program output that a bug had occurred. The user realized only through a DIDUCE invariant violation that a bug had occurred - otherwise he would have continued to use the erroneous output produced by the buggy program.

Here's a brief summary of how DIDUCE works:

Step 1: Instrumentation: DIDUCE instruments the underlying classes so it can check for violations. This happens transparently “under the hood”.

Step 2: Training: The instrumented program is run on test cases for which the program works.

Step 3: Invariant checking: The instrumented program is run on test cases for which the program fails.

DIDUCE and its applications are discussed in an ICSE 2002 Paper, entitled "Tracking Down Software Bugs Using Automatic Anomaly Detection" which is available at the following URL:

http://suif.stanford.edu/papers/Diduce.pdf

DIDUCE is covered by the GNU General Public License (GPL). Please refer to http://www.gnu.org/copyleft/gpl.html for details.

2. Getting Started

There are three ways of running DIDUCE on your application. These are explained below.

1. The fastest way of getting started with DIDUCE is to set the JAVA_HOME environment variable to the directory where J2SE version 1.4 or later is installed.

UNIX:

> setenv JAVA_HOME <j2se-home>

> ${DIDUCE_HOME}/bin/**djava-train** <program name> <passing args>

This will train DIDUCE using the instrumented program and writes out the invariants learned to a file named program.inv.

> ${DIDUCE_HOME}/bin/**djava-run** <program name> <failing args>

Uses the invariants generated during training for checking invariants on the inputs for which the program does not work. This will check invariant violations as the program runs. A GUI is started that allows you to browse invariants and violations, correlated with the source code, even as the program is running.

2. The second way of running DIDUCE is to use diduce.run – this method provides more control to the user.

Step 1: Set your classpath as you usually would in order to run the program. Add the files diduce.jar and BCEL.jar (from the distribution) to your classpath. e.g.

```
setenv CLASSPATH <usual classpath>
setenv DIDUCE <full path to directory where you untar'ed the distribution>
setenv CLASSPATH $DIDUCE/diduce.jar:$DIDUCE/BCEL.jar:$CLASSPATH
```

Be careful not to include diduce-cache.jar in the CLASSPATH at this stage.

Step 2: You will need to set the system class loader property to diduce.DiduceClassLoader that will do on-the-fly instrumentation of your application classes as and when they are loaded. To train DIDUCE using instrumented program use:

```
java -Djava.system.class.loader=diduce.DiduceClassLoader  diduce.run -r -o
program.inv <program name> <program arguments>
```

This writes out the invariants learned to a file program.inv.

Step3: Now add diduce-cache.jar that contains instrumented files to the beginning of the classpath. e.g:

```
setenv CLASSPATH diduce-cache.jar:$CLASSPATH
```

This will make sure that, for the classes you chose to train, the instrumented ones will be picked up during the checking runs. You still want your original class path to be present in order to get to the classes which you did not instrument (if any).

To run further training test cases beyond the first one:

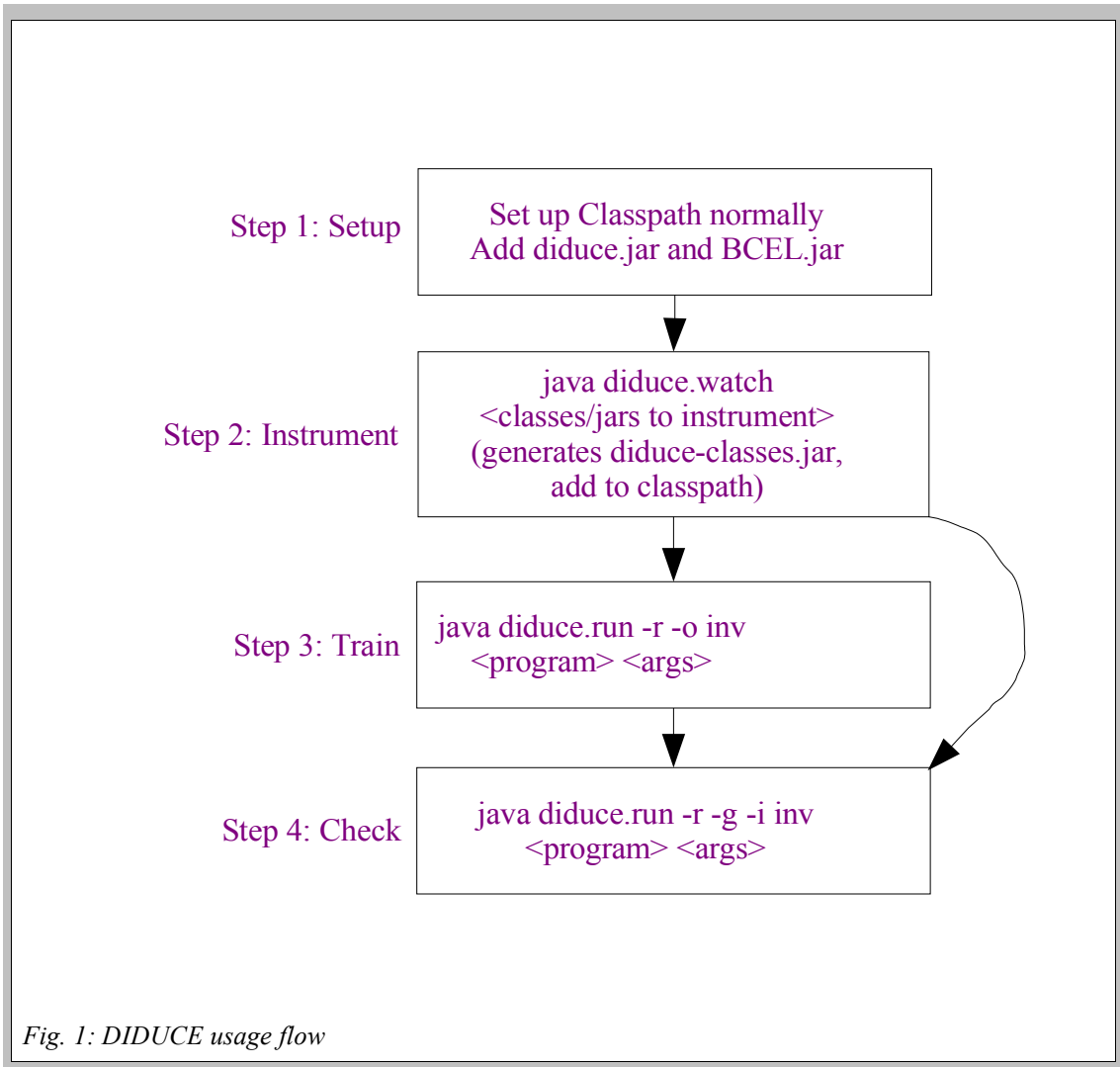

Fig. 1: DIDUCE usage flow

```
java diduce.run -i program.inv -o program.inv -r <program name> <program arguments>
```

This command will read the program.inv generated by the previous test case, build on top of it, and write out the invariants learnt to the same file. Note that the directory in which you run the test inputs need not be the same as the directory in which the instrumented files were generated.

Step 4: Use the invariants detected in step 3 for checking invariants on the inputs for which the program does not work. Now, run the following command:

```
java  diduce.run -i program.inv -r -g <program name> <program arguments>
```

This will check invariant violations as the program runs. If there are a lot of messages, you should ignore the first few, and look at the ones which are closest to the point of failure. With the -g option, a GUI is started that allows you to browse invariants and violations, correlated with the source code, even as the program is running.

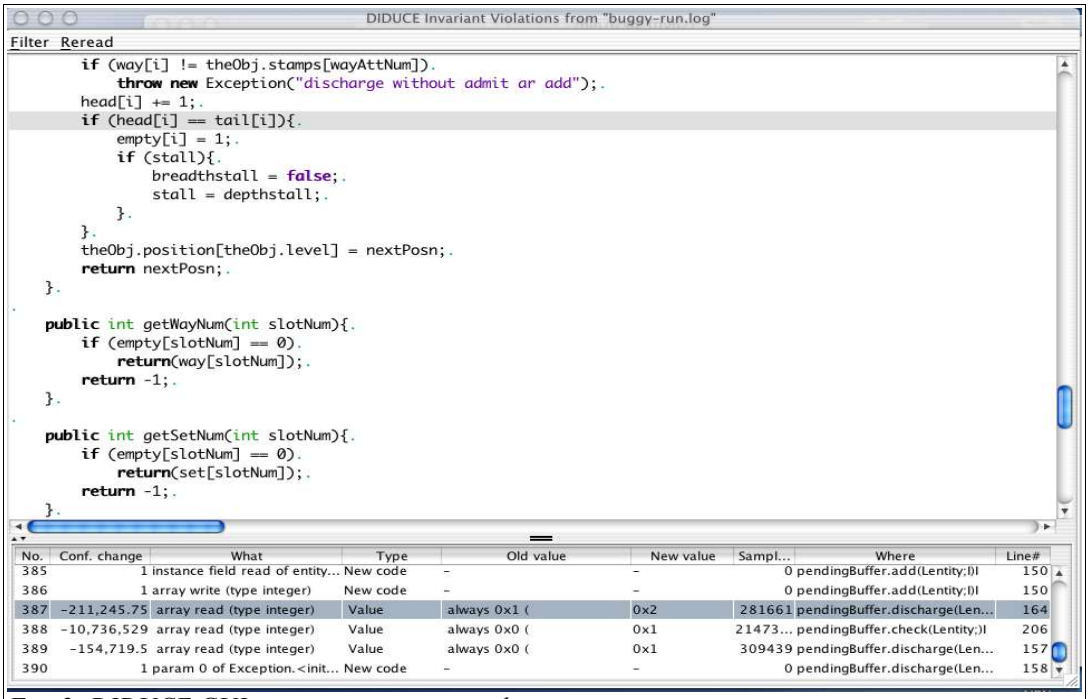

Fig. 2: DIDUCE GUI reporting invariant violations

That's it!

Note: In this version of DIDUCE, on-the-fly instrumentation of J2SE library classes is not possible. If you want DIDUCE to instrument those classes use the third way of running diduce.

3. Third way of running diduce is to do instrumentation on your own using diduce. Then use diduce.run to run your program.

Below are the steps involved in it. Also see figure 1 below.

Step 1: Set up your classpath. Same as step 1 above.

Step 2: Decide on a list of classes and jar files to watch, and instrument them:

```
java diduce.watch <list of all classes/jar files which you want to watch behaviour for>
```

Class names can be either with specified with '/' or '.' for package delimiters, and can optionally end with the extension .class. Jar files can also be specified, in which case each class file in the jar file is instrumented. For example:

```
java diduce.watch foo a.b.c x/y/z.class p/q/r mypack.jar
```

will instrument the classes 'foo', 'a.b.c', 'x.y.z', 'p.q.r' and all the classes in mypack.jar, and write the instrumented versions to the file diduce-cache.jar. The individual classes (e.g. a.b.c) can exist inside other jar files on your classpath. Specifying a jar file to diduce.watch is simply a convenience mechanism for when all files in the .jar have to be instrumented so you don't have to list each class in the jar file. Other files such as manifests and property files which may be present in the jar file are copied to diduce-cache.jar without change.

Note: All classes which you intend to instrument for a given program must be instrumented within one invocation of diduce.watch, i.e. you cannot instrument a few files, then instrument some more later, etc. (If this is a serious problem, please let us know as a request for enhancement)

As a side effect, a file diduce.watch.out is created in the current directory with details about each instrumentation point.

Step 3: Now add diduce-cache.jar that contains instrumented files to the beginning of the classpath.

(Note: if you have instrumented java.* or javax.* files, you'll need to prepend diduce-cache.jar to your bootclasspath instead) Now train DIDUCE using the instrumented program:

```
java diduce.run -r -o program.inv <program name> <program arguments>
```

This writes out the invariants learned to a file program.inv. To run further training test cases beyond the first one:

```
java diduce.run -i program.inv -o program.inv -r <program name> <program arguments>
```

Step 4: Check invariant violations. Same as Step 4 above.

That's it!

How DIDUCE Works

While the procedure described in the previous section is adequate to begin using DIDUCE, it helps to understand (and even control) what invariants DIDUCE tracks and checks.

Instrumented Program Points

Invariants are maintained at each of the following types of program points:

- Method calls and return values
- Accesses to instance and class fields of objects (including arrays)

Note that local variables in a method are not watched.

Invariants Tested

At each program point described above, DIDUCE tracks invariants on a set of "tracked expressions". The default set of tracked expressions consists of:

- the value of the variable being read or written. For procedure call sites, there is one tracked expression per parameter and one tracked expression for the return value (unless it's a void method).
- the change in value of a variable when it is written
- the runtime type of the base object which is being read from or written to

You can also add your own invariants for detection and testing, which may make sense in your application domain or in your program. See the next chapter for how to specify your own invariants, and what invariants are available in the DIDUCE invariant library.

Reading an Invariant Violation Message

There are 3 kinds of messages which DIDUCE will output in checking mode. The messages in the GUI are contain essentially the same information - they are just reformatted for easy display, browsing and sorting. (The precise format of such messages may change from version to version or between text and GUI versions)

1) New code executed

```
DIDUCE: New code executed: read of object array in method Sanity.foo, line 141
```

This means the code at line 141 which causes a read to an object array was executed for the first time during all the training runs and the checking run (thus far)

2) New value for a variable when it was constant in the past

```
DIDUCE: (Confidence change: -23) Invariant violation in write of object field
directory.dirNum (type integer) in directory.<init>, line 23: change in value on 2
previous occasions was always 0x0, now it is 0x12
```

This indicates that a value which was always a constant in the past (over all training runs, and up to this point in the checking run) is now acquiring a new value.

Note that a "value" in this case may be either the value read or written to a variable, or the amount of change to the value of a variable (on a write) - as in the example above - or the runtime type of a variable.

3) Similarly, DIDUCE prints out a message when it knows that a variable attains a new value which it had never seen in it's history (over the training runs, and thus far in the checking run)

```
DIDUCE: (Confidence change: -10.2) Invariant violation in read of object field
entity.address (type long) in method directory.initEntry, line 76: value on 8 previous
occasions matched the value 0x5 in bits 0x3, now it is 0x8
```

This is the hardest kind of invariant violation to read. It implies the following things:

On the first execution of this code fragment, the value of the tracked expression was 0x5.

On all subsequent executions of this code fragment, the bits set in the number 0x3 (i.e. the last 2 bits) matched those same bits from the first-seen value (0x5). This implies that the last 2 bits for this particular value were always '01'. However, there is now a value at this program point, which does not match that pattern. e.g. in the above case the value is 0x8 which does not have '01' in its last 2 bits, and hence DIDUCE is reporting an invariant violation. Once again the message will tell you whether it is talking about the actual value of the element read or written, or the value change of the element (from its existing value).

(If you find matching up bits to be confusing, don't worry, you're not the only one! Just take it to mean that the value seen at the program point is different from that in the history of the program)

In all cases, look for confidence changes of the highest magnitude first. A higher magnitude confidence change means an invariant which DIDUCE had strong confidence in (based on the history of the program) is being violated, i.e. something which was true very frequently in the past is no longer true.

Using the GUI and Source Browser

You can use the DIDUCE GUI to browse through invariant violations (even as the instrumented program is running). To do this, just start the checking run with the -g option:

```
java -Ddiduce.sp=<source path> diduce.run -g -r -v <program> <program args>
```

or save the results of the run to a file, and then run diduce.show.

```
java  diduce.run -r -v -o program.inv <program> <program.args>
java -Ddiduce.sp=<source path> diduce.show program.inv
```

If you use the diduce.sp property to specify source paths (multiple paths separated by ':', like CLASSPATH) the GUI will also throw up a source code browser with the relevant line from your source file highlighted. If you are viewing the invariant violations as the program is running, and want to update the display just press Control-R. The filter menu helps in browsing through a large number of messages: Control-B hides messages before the currently highlighted one and Control-C hides messages below a certain confidence change level. Control-E displays invariants or violations associated with the selected text in the source window. Control-D disables display of the type of invariant in the currently selected row. Control-A resets all filtering and redisplays all messages.

Within the GUI, you can click on the name of a column to sort by that column in ascending order (shift-click for descending order). This is useful, for example, to see the invariant violations with the highest confidence changes. You can also move and resize the columns if you wish.

The GUI window allows for viewing both invariant relaxations and the final invariants learned at the end of a run.

Line Numbering Information

Line numbers reported by DIDUCE always refers to the line numbers in the original Java source file. If line numbers are reported as 0, it means that line number information is unavailable in the source file. You can compile with the -g option to javac (or the equivalent switch in other compilers) to make sure that line number information is present in the file.

When using DIDUCE with class files for which you don't have the source code, a good strategy is to use a decompiler like 'jad' (http://kpdus.tripod.com/jad.html). However, the line numbers printed by DIDUCE may not correlate exactly with the line numbering generated by the decompiler (depending on the decompiler - jad doesn't preserve correct line numbering for example.) In such cases, it's useful to look at the function name, and make an approximate guess depending on the information DIDUCE prints out for surrounding lines, etc.

Long-running programs

If you have programs on which you do not have an adequate set of test cases which pass, you can still use DIDUCE to debug errors which might occur on long running executions. The idea here is to jump directly to step 4, and bypass the training in step 3 altogether. Step 4 will then effectively start with no training, and will therefore will encounter a large number of initial invariant violations as it learns about the program. However, if the program runs long enough, DIDUCE will eventually learn everything about the program during the course of this run itself, and will mostly stop reporting violations except for the real corner cases in the execution. You should then examine only the invariant violation messages towards the end of the run.

In this case, since there is no training file, you will omit the "-i program.inv" option from step 4, and run:

```
java diduce.run -r -g <program name> <program arguments>
```

The final state of invariants detected is also saved to the output file (if any) at the end of the training or checking run. You can use diduce.show to view this output file.

Instrumentation overhead:

The instrumentation overhead of DIDUCE can be fairly high depending on what part of the application you decide to instrument, the runtime execution percentage of that part, etc. If you instrument all your classes, the slowdown is likely be about 20-30X.

For this reason, it's best if you have an idea of which module or which classes might have a bug, and choose to instrument only those. If you cannot do this, or if the instrumentation overhead is still too high, a good strategy is to break the list of classes you want to instrument into different groups, instrument them independently and run them all in parallel on different machines. Since each instrumented class observes and checks behaviour only on code inside that class (and not on interactions with external classes), this kind of instrumentation generates exactly the same information as if you had instrumented all classes at once.

Instrumenting system classes

Instrumenting core system classes (some of the java.* classes) mostly will NOT work because some JVMs make assumptions about some standard system classes and cannot deal with an instrumented system class. Therefore, you have to leave these out of your set of instrumented classes. (if instrumenting system classes somehow works for you, please let me know). javax.* classes are usually ok to instrument. Remember that from Java 2SE SDK1.2 onwards (at least on Sun JDK's), the classpath option doesn't control system files; the -Xbootclasspath option does.

3. DIDUCE Options

There are 2 user-visible parts to DIDUCE: the instrumentation system (invoked with diduce.watch) and the runtime system (invoked with diduce.run).

Instrumentation Options

The DIDUCE instrumentation system generates instrumented version of the user's classes which can be used by the DIDUCE runtime system for detecting and checking invariants. DIDUCE instruments the users classes according to optional constraints and writes the resulting classes to a file called a class cache, which is a single jar file. Inserting this jar file at the beginning of the CLASSPATH, leads the Java runtime to use the instrumented classes instead of the original ones.

When dealing with a large program, it is useful to incrementally re-instrument only those classes which have changed. DIDUCE supports this via a class cache; if the instrumented version of the class the user is trying to instrument is already in the supplied class cache, it is used directly , otherwise the class is instrumented afresh. Only one class cache can be supplied at a time. In case instrumentation constraints were specified when creating the original cache file, the same constraints apply to any newly instrumented class as well. In this case, any instrumentation constraints specified on the command line are silently ignored. (In other words, the -p and -c options should not be used simultaneously to diduce.watch).

The following properties or switches can be specified to diduce.watch.

-c <filename>

-Ddiduce.cache=<filename>

Uses the given file as an input class cache, and re-instruments only the classes which have changed.

-o <filename>

-Ddiduce.output=<filename>

Instrumented classes are output to this jar file. Default is diduce-cache.jar.

-p <filename>

-Ddiduce.control.file=<filename>

reads XML constraints on the instrumentation (see below)

-l <filename>

-Ddiduce.watch.log=<filename>

Detailed log messages go to this file. Default is diduce.watch.log

Runtime Options

The following properties can be specified while running a program under DIDUCE. Most options can be specified either with an argument to diduce.run, or by setting a java property.

-i <filename>

-Ddiduce.read=<filename>

reads invariants from a file at startup

-o <filename>

-Ddiduce.write=<filename>

writes learned invariants (as well as invariant relaxations) to a file at the end of the program run. (Also needs -r, see below)

-r

-Ddiduce.report=1

Checks for invariant violations. (**Currently, this is required for the -o option to work.)** In the future, you can decouple saving invariants from saving invariant relaxations)

-t

-Ddiduce.timestamp=1

prints out the time at which every diduce message is printed - useful for long running programs or server/daemon type programs.

-nc <val>

-Ddiduce.new.code.confidence.change=<val>

This option controls the confidence change reported for code which is executed for the first time. The default value is 100. Higher values cause newly executed code to be reported with greater prominence.

-Ddiduce.dir=<directory>

This option specifies a temporary directory name where the instrumented classes are stored (before being jar'ed up in the diduce-cache.jar file). The instrumented class files are always jar'ed and then deleted, so this option rarely needs to be specified. The default directory name is ./instrumented.

-g

-Ddiduce.gui

This option specifies that a GUI should be started to view the invariant relaxations, as well as learned invariants even as the program is running. Remember to refresh the GUI (Ctrl-R) periodically to update to the latest set of invariants and relaxations.

-sp

-Ddiduce.sp=<path1>:<path2>:...

This option specifies the list of paths to lookup to find source code in the source browser (with the -g option to diduce.run, or with diduce.show). The source path is specified similarly to the class path - as a ':' separated list of paths. Starting from each path component, the full package name and class are searched. (Inner classes are supported by removing the inner class name after $)

You would specify these properties on the java command line, e.g.

```
java -Ddiduce.timestamp=1 diduce.run -g -v -r my_program my_args
```

Specifying Classes of Program Points

DIDUCE allows the user to specify a set of program points in order to control some aspect of its behavior at those program points. This is done by specifying:

```
java -Ddiduce.control.file=<file> diduce.watch  ...
```

where <file> contains program points specifed in an XML format, whose elements are described below.

The attribute value of the XML elements can have a '*' character to indicate a wildcard match. A program point has to match each field in the specification in order to generate a match. Attributes not specified are considered the same as '*' and hence not compared. All matching (both on attribute names and values) is case-sensitive.

The following sections describe the XML elements used to specify the control information:

***ControlInfo** tag*

This is the outermost XML structure containing a sequence of one or more control specifications i.e, sequence of tags that control how the specified program point/s are to be treated. For example the tags *Watch* and *DoNotWatch* control the enabling and disabling the instrumentation of program points.

ProgramPoint tag

Describes a program point. It consist of all the elements and attributes described below that represent a program point.

Elements of ***ProgramPoint** tag**:***

Method tag

Describes the method in which program point lies. The method is specified by defining one or more of these three attributes:

name **params** **returns**

The 'name' attribute is the fully qualified name of the method (along with the class and package name).

The 'param' attribute is the list of parameters to the method seperated by comma.

The 'returns' attribute is the type that the method returns.

Example: The method below:

```
Class A {abstract void bar(int i, String str);}
```

would have the specification:

```
<Method
name="A.bar"     params="int,java.lang.String"     returns="void"
></Method>
```

Wildcard matching of methods:

The attribute values can contain '*' in them for wildcard matching.

As mentioned before, absence of any attribute is equivalent to specifying it's value as '*'

Target tag

The name of the field being accessed or the method being called. This must be the field or method name which is present in the class file (which is the compile time type, and may not be the same as the runtime type.). The field name is specified by the attribute:

field

The field attribute value is the fully qualified name of the field (along with the class and package name).

Example: The target field Pkg.C.foo and the target method:

```
Class C {abstract String foo (int x, Object o);}
```

would have the specification:

```
<Target field="Pkg.C.foo"> </Target>
```

and

```
<Target>
  <Method name="C.foo" params="int,java.lang.Object"
          returns="java.lang.String">
  </Method>
</Target>
```

Attributes of *ProgramPoint*

type=<"int" | "char" | "byte" | "boolean" | "short" | "long" | "float" | "double" | "object"

|"array">

The Java type of the value being accessed at the program point.

write="true" | "false"

Matches only writes (need something to match only reads ??)

static="true" | "false"

Matches only static fields. (Note: Only for fields, does not apply to static methods)

num=<#>

The parameter number of the target method. (Check: How are params for virtual methods numbered – from 0 or 1 ?)

line=<#>

Matches only program points which match the given line number in the source file. Line number information has to be present in the class file for this to work, otherwise DIDUCE will consider all points to have a line number attribute of 0. To ensure line number information is generated in the class file, use the -g option to javac.

op=<"param" | "retval" | "field" | "array">

Selects the kind of program point. Array refers to array (load or store) instructions, field refers to field accesses (both static and instance fields), retval refers to the return value of procedures, and param refers to the parameter of a procedures (the precise parameter number is given by num).

Enabling/Disabling Instrumentation

By default, all program points described earlier except those which access words of type *floating point* or *double* are instrumented. However, instrumentation can be controlled by specifying to diduce.watch a property "diduce.props", which points to a XML file containing the control information.The tags below disable or enable the instrumentation of program points:

DoNotWatch tag

Describes the program point/s *not* to be watched (instrumented) for. By default all the program points are instrumented (except those that access float or double), unless they are specified using this tag.

Watch tag

Describes the program point/s to be watched for. This tag is useful for selectively adding back the program points that were excluded using **DoNotWatch** tag.

The property "diduce.props" must be specified at instrumentation time, i.e. to diduce.watch, not at run time (to diduce.run). The Watch/DoNotWatch enable/disable instrumentation for all program points which match the associated spec. The same program point can match multiple specs- if this happens, the last directive which matches a program point is the final word on whether the program point is instrumented or not.

Examples

- Disable all accesses to field Pkg.C.junk:

```
<?xml version="1.0" encoding="UTF-8"?>
<ControlInfo>
    <DoNotWatch>
        <ProgramPoint>
            <Target field="Pkg.C.junk"></Target>
        </ProgramPoint>
    </DoNotWatch>
</ControlInfo>
```

- Disable all instrumentation in C.hot_method:

```
<?xml version="1.0" encoding="UTF-8"?>
<ControlInfo>
    <DoNotWatch>
```

```
        <ProgramPoint>
            <Method name=”Pkg.C.hot_method”> </Method>
        </ProgramPoint>
    </DoNotWatch>
</ControlInfo>
```

- Disable all instrumentation in Pkg.C.hot_method, except for writes to Pkg.D.x

```
<?xml version="1.0" encoding="UTF-8"?>
<ControlInfo>
    <DoNotWatch>
        <ProgramPoint>
            <Method> name="Pkg.C.hot_method"></Method>
        </ProgramPoint>
    </DoNotWatch>
    <Watch> <ProgramPoint> write=”true”>
              <Target field=”Pkg.D.x”> </Target>
            </ProgramPoint>
    </Watch>
</ControlInfo>
```

- Disable instrumentation of array reads in Pkg.C.hot_method

```
<?xml version="1.0" encoding="UTF-8"?>
<ControlInfo>
      <DoNotWatch>
          <ProgramPoint> op=”array” write=”false”>
                <Method name=”Pkg.C.hot_method”> </Method>
          </ProgramPoint>
      </DoNotWatch>
</ControlInfo>
```

- Disable instrumentation on param 1 and return value of calls to Pkg.C.foo in Pkg.C.hot_method:

```
<?xml version="1.0" encoding="UTF-8"?>
<ControlInfo>
      <DoNotWatch
          <ProgramPoint> op=”param” num=”1”>
```

```
            <Target> <Method name="Pkg.C.foo"> </Method> </Target>
            <Method name="Pkg.C.hot_method"> </Method>
        </ProgramPoint>
    </DoNotWatch>
</ControlInfo>
```

(NOTE for section below: User-specified helpers are temporarily disabled !)

Actually, the instrumentation selection is more powerful than shown above. A class name can be used in place of the tags "Watch" and "DoNotWatch". This class must extend diduce.Runtime and provide at least all the functions which will be invoked at the program points which the spec applies to. e.g.

```
<?xml version="1.0" encoding="UTF-8"?>
<ControlInfo>
 <SpecialRuntime op="array" write="true" type="ref"></SpecialRuntime>
</ControlInfo>
```

will call SpecialRuntime.register_object_ref_store instead of calling the usual Runtime.register_object_ref_store. The SpecialRuntime class needs to contain only the methods it really needs, i.e. if SpecialRuntime will be used only at points where type=ref, it need not provide methods to handle float, etc. If SpecialRuntime does not want to provide all methods, even for points which it applies to, it can omit the definitions of functions it is not interested in overriding - they are inherited from diduce.Runtime.

The user needs to make sure that SpecialHelper is available in the classpath, etc at the time of running the instrumented program, and has all the needed methods etc, otherwise will see exceptions like NoClassFound or IncompatibleClassChange.

The Watch tag simply maps to the default helper (called diduce.Helper). The DoNotWatch tag causes no instrumentation to be inserted at all.

Equivalent Program Points

Sometimes it is useful to maintain a single invariant on tracked expressions across a set of related program points, instead of at each program point. For example, a user may like to know when field F of any object of class C is assigned a value > 100 across the entire program, regardless of which program point performs the assignment.

To enable this, the user groups multiple program points together using a syntax similar to that used in the Watch/DoNotWatch structures. The tag for marking multiple kinds of program points equivalent for the purpose of invariant detection is (duh) "Equivalent".

```
<Equivalent>
 <Watch> <ProgramPoint> ... program point1... </ProgramPoint></Watch>
 <Watch> <ProgramPoint> ... program point2... </ProgramPoint></Watch>
</Equivalent>
      :
```

With this, all program points matching the specifications spec1 and spec2 are assigned a single set of invariants. For example, the following specification uses a single invariant to track all writes to Foo.bar, regardless of where they are in the program.

```
<?xml version="1.0" encoding="UTF-8"?>
<ControlInfo>
    <Equivalent>
        <Watch> <ProgramPoint>
            <Target field=”Foo.bar”> </Target>
        </ProgramPoint> </Watch>
        <Watch write=”true”> </Watch>
    <Equivalent>
</ControlInfo>
```

Note that the set of tracked expressions assigned to each program point (see below for how this is done) **must** be the same. For example, all writes to Foo.bar in the above example must be assigned the same invariant set, or the results are undefined.

(Aside: Actually, the set of tracked expressions assigned to this set of program point is guaranteed to be that specified for the first program point executed in the run. However, merging invariants from different runs etc may not be possible because each run may have maintain a different set of tracked expressions for this set of program points)

Not: = feature not extensively tested. Extensive error checking definitely not done!

4. Invariant Selection

This section describes which invariants are tracked for each type of program point, and how users can control the defaults.

Invariant Library

DIDUCE has a library of predefined invariants. The following invariants are currently available:

IntValInvariant: invariant on bits of the values at an integer access

IntValRangeInvariant: invariant on the range of values taken by an integer access

IntDiffInvariant: for integer stores, invariant on bits of difference between old and new values

IntDiffRangeInvariant: for integer stores, tracks range of the difference between old and new values.

Values of type boolean, byte, char, short are promoted to integer for the purpose of invariant detection. Values of long type have long versions of the integer invariants. Currently, there are no float or double type invariants.

ArrayLengthInvariant: invariant on array length at array access points.

Type Invariants:

TypeTrackingInvariant: invariant on the type of value accessed (when the value is an object)

BaseTypeInvariant: invariant on the base type of the object reference whose field is accessed

TypeDiffInvariant: On an object store, did the type of the object in the stored location change ?

SelfLoopInvariant: For an object field access, can the accessed field point to the parent object ?

In addition, users can write their own invariants by extending the class diduce.Invariant.

Specifying Custom Invariants

<TBD: Explain how users write their own invariants as a subclass of diduce.Invariant, and the mechanism of the control file. Meanwhile please read the code in diduce.Invariants - it's quite straightforward>

(Note for Stanford people: if you write your own invariants which might be generally applicable, please feed them back into the CVS tree)

Notes section: v0.9 does not acquire locks on SPPdata before updating it.

User Control of Invariants

Which invariants are associated with specific program points can be controlled by using the property diduce.control.file=<file> to diduce.run. If the control file exists, it is used to generate the define the set of invariants, otherwise the default invariants listed in the next section are used.

The format for invariant specification is in XML. The invariants are specified using the ***Invariant*** tag. The elements of invariant tag include a list of invariants and the corresponding program point. The next section has an example of XML specifification for default invariants being used.

Default Invariants

These are the invariants tracked by default on the corresponding categories of program points.

<ControlInfo>

<Invariant>

<**diduce.IntValInvariant**></diduce.IntValInvariant>

<ProgramPoint **type="int|byte|short|char|boolean" write="false"**> </ProgramPoint>

</Invariant>

<Invariant>

<**diduce.IntValInvariant**> </diduce.IntValInvariant>

<**diduce.IntDiffInvariant**> </diduce.IntDiffInvariant>

<ProgramPoint **type="int|byte|short|char|boolean" write="true "**> </ProgramPoint>

</Invariant>

<Invariant>

<**diduce.IntValInvariant**> </diduce.IntValInvariant>

<**diduce.BaseTypeInvariant>** </diduce.BaseTypeInvariant>

<ProgramPoint **op="field" type="int|byte|short|char|boolean" static="false" write="false"**> </ProgramPoint>

</Invariant>

<Invariant>

<diduce.LongValInvariant> </diduce.LongValInvariant>

<ProgramPoint **type="long" write="false"**> <ProgramPoint >

</Invariant>

<Invariant>

<**diduce.LongValInvariant**> </diduce.LongValInvariant>

<**diduce.LongDiffInvariant>** </diduce.LongDiffInvariant>

<ProgramPoint **type="long" write="true"**> </ProgramPoint>

</Invariant>

<Invariant>

<diduce.LongValInvariant> </diduce.LongValInvariant>

<diduce.BaseTypeInvariant> </diduce.BaseTypeInvariant>

<ProgramPoint **op="field" type="long" static="false" write="false"**>

</ProgramPoint>

</Invariant>

<Invariant>

<diduce.LongValInvariant> </diduce.LongValInvariant>

<diduce.BaseTypeInvariant> </diduce.BaseTypeInvariant>

<diduce.LongDiffInvariant> </ diduce.LongDiffInvariant>

<ProgramPoint> **op="field" type="long" static="false" write="true"**> </ProgramPoint>
</Invariant>

<Invariant>

<diduce.TypeInvariant> </diduce.TypeInvariant>

<ProgramPoint **type="object|array" write="false"**> </ProgramPoint>

</Invariant>
<Invariant>

<diduce.TypeInvariant> </diduce.TypeInvariant>

<diduce.TypeChangeInvariant> </diduce.TypeChangeInvariant>

<ProgramPoint **<type="object|array" write="true"**> </ProgramPoint>

</Invariant>

<Invariant>

<diduce.TypeInvariant> </diduce.TypeInvariant>

<diduce.BaseTypeInvariant> </diduce.BaseTypeInvariant>

<ProgramPoint> op="field" type="object|array" static="false"
write="false" </ProgramPoint>

</Invariant>
<Invariant>

<diduce.TypeInvariant </diduce.TypeInvariant >

<diduce.TypeChangeInvariant> </diduce.TypeChangeInvariant >

<diduce.BaseTypeInvariant> </diduce.BaseTypeInvariant>

<ProgramPoint> op="field" type="object|array" static="false" write="true">
</ProgramPoint>

</Invariant>

</ControlInfo>

Reusing XML definitions:

The ***programPoint*** definations are likely to be re-used in the control file specification. XML specification allows re-usage of structures using Entities. Here is an example for reusing a program point in the specification for:

Disable all instrumentation in Pkg.C.hot_method, except for writes to Pkg.D.x.

Associate `mypackage.FieldWriteInvariant` with the writes to Pkg.D.x.

The program point description for "writes to Pkg.D.x" can be declared once using XML Entities

and it can be reused later.

```
<?xml version="1.0" encoding="UTF-8"?>
  <!DOCTYPE ControlInfo
  [
   <!ENTITY p "<ProgramPoint write='true'>
                    <Target field='Pkg.D.x'> </Target>
              </ProgramPoint>”>
  ]>
  <ControlInfo>
      <DoNotWatch>
          <ProgramPoint>
              <Method> name="Pkg.C.hot_method"></Method>
          </ProgramPoint>
      </DoNotWatch>
      <Watch> &p </Watch>
       <Invariant
            <mypackage.FieldWriteInvariant>
           </mypackage.FieldWriteInvarint>
            &p
       </Invariant>
  </ControlInfo>
```

Applications of DIDUCE

This section outlines some scenarios in which DIDUCE may be useful.

Debugging Regression Failures

It is a common occurrence that a program that works correctly on many inputs, fails on others. DIDUCE can be used to quickly pinpoint differences in behavior between the successful and the failing runs. For example, DIDUCE could be used to automatically provide debug information upon detection of failures in a nightly regression run, by first extracting invariants from test cases that pass, and checking for invariant violations on the failing cases. The list of invariant violations can then be presented in the morning along with the failing tests to a test engineer or developer to reduce debugging time.

Debugging Long-running Programs

When bugs which are encountered only after the program has been running for a long time, a developer typically guesses what the problem is and tries to gain visibility on the key variables or code segments by adding debugging statements, assertions, and breakpoints into the program. This trial-and-error process can be time consuming for long-running programs. Moreover, the programmer's own intuitions may not necessarily be dependable, since the errors may be caused by his own misconceptions in the first place. DIDUCE blindly and continually monitors all the variables in the program and therefore has a good chance at reporting the anomalous event causing the bug. For long running programs, a training set may not even be necessary. Assuming there are no bugs in the early part of the run, DIDUCE can flag anomalous behavior in the later parts of the run. This method is often useful for simulators or other programs for which the correct output is not known.

Supporting Code Evolution

Another important use for DIDUCE is to check if modifications to part of the program unexpectedly alter the behavior of other parts. For example, while checking in code, developers can simply instrument the parts of the program whose behaviour they believe remains unaltered. Then they train DIDUCE on the original program, and check the invariants learnt on the program after the update. This is especially useful in assisting new programmers to ensure that changes they make do not break invariant assumptions in the rest of the code.

Debugging Component Based Software

DIDUCE can be used in the bring-up of a component-based system. Normally, a program must run correctly on some inputs, or for some duration, before DIDUCE can successfully extract meaningful invariants for the program. For component-based software, however, we can first train DIDUCE on other codes that use the same components correctly, and apply it to check the behavior of the component in the context of the new software.

Random Testing

Producing test cases for a program can be quite tedious because the expected results must also be prepared for comparison with the program's output. Assuming there are some tests for which the results are known, we first train DIDUCE on these tests, and use the invariants gathered to check the runs on inputs with no known outputs. With this approach, it is possible to test software with, say, pseudo-random inputs for which no answers are known a priori. Invariant violations detected indicate bugs in the program, or at least expose corner cases which did not occur during directed testing.

5. FAQ

To be created

6. Release History

Release Notes for v1.0

Automatic Instrumentation at class loading time

Release Notes for v0.9.7

Class Cache added, made robust.

XML format used for instrumentation constraints.

Release Notes for v0.9.5

GUI is now integrated into diduce.run itself, so users can browse relaxations and invariants as the program is running.

Invariants and violations are stored in the same invariants database (a single file).

Command line switches added to diduce.run (-o, -i, -g, -r, -v)

Release Notes for v0.9

DIDUCE is now much more customizable, i.e. Users can specify their own helper's, invariants, etc.

Much cleanup in the code for specifying program points

Added new invariant: SelfLoopInvariant

Changed property name diduce.verbose to diduce.check

Release notes for v0.7

GUI improved to allow filtering and rereading log files.

Release notes for v0.6

This is a relatively major upgrade, that's why the jump from 0.4 to 0.6

New feature: Ability to specify what gets instrumented and what doesn't

New feature: Printing class names on type invariants

New feature: gui (diduce.show)

Source code reorganized and cleaned up

SPP descriptor format cleaned up

Release notes for v0.4

some bug fixes, no change in functionality

Release notes for v0.3

1. DIDUCE currently will not be able to train correctly on program runs which end with a call to System.exit (Throwing an uncaught exception or returning from main is fine). Therefore your program ideally should not call System.exit directly. Similarly, you must ensure that the training run completes normally, because DIDUCE will not be able to write out partial training file if you terminate the training run with Ctrl-C.

2. In rev. 0.4, we also monitor invariants on the return values of a program.

In rev. 0.5 we will have invariants on parameters to procedures.

Release notes for v0.2

DIDUCE started as a CS343 course project at Stanford University under Monica Lam.