Jlint – status of version 3.0

Raphael Ackermann raphy@student.ethz.ch

June 9, 2004

Contents

1 Introduction **3**

2 Test Framework **3**

3 Adding New Test Cases **6**

4 Found Errors, Bug Fixes **6**

4.1 try–catch–finally constructs in Java 1.3 and Java 1.4 6

4.2 Context Handling in `Jlint` 8

4.3 Bug Related Errors 9

4.4 Minimal class file that reproduced the bug 9

4.5 Bug fix 10

5 Changes to the Build Process **11**

6 Results and Conclusion **12**

1 Introduction

`jlint` is a static model checker for java programs.
It can be found at the following URL's: http://www.artho.com/jlint/
http://www.sf.org/jlint/

The development activity of `jlint` has been very low in the last two years. At the same time a new version of the gnu compiler g++ 3.X and a new java version 1.4 became quite popular. Java 1.4 introduced some changes in the way java programs were compiled. These changes caused a crash in `jlint` when analysing certain class files. Furthermore, jlint's sources couldn't be be compiled using the new g++ 3.x compiler and therefore in a first step the sources had to be changed to allow compilation using a new gnu compiler. In a second step the bug was to be fixed.

There were also a couple of other things which suggested to revise jlint and to do a new major release with an up to date jlint. This meant writing the hitherto missing ./configure script, merging some patches, making jlint able to work on 64 bit architectures, and eliminating the warnings of valgrind.

Such a long time of almost no development is of course not wanted and a broader basis and more active development on jlint was looked for. It is hoped that this goal will be reached by moving the CVS repositories and the web page to www.sourceforge.net where everybody can contribute to the further development of jlint. While fixing the bug, a test framework was developed which supports regression testing on new versions and different platforms. Unit Testing could be added in a future release.

2 Test Framework

A test framework allows you to control the changes made in the source code during development. Such a framework can consist of several kind of tests.

The one test pattern we will look at here is regression tests. In a regression test you need to have a predefined behaviour and output of your program before actually running any tests. The goal is then that on every platform and for every future version of the program the output matches the specified output and that the program works in a predefined manner.

Testing on different platforms is important because e.g., Macintosh computers use a different memory alignment than windows IA32 architectures.

Running the test framework after every change in the source code and checking if the output still matches the original output is a good way to find an error or to find out whether a change in the sources which seems to be correct has any side effects on another part of the program.

There are different kind of regression tests possible. Two of them are

black box testing and white box testing.

Black box testing as it was added to `jlint` in this semester project works roughly like this:

You need a number of different inputs, the test cases. A correct version of the program is then used to create the predefined outputs and indicated above. The inputs should never change, once they do, you will have to create new outputs. The created output files are saved and are referred to as reference outputs. They specify the desired output which should be matched by future versions of your program.

```
create test framework              run test framework


          input                              input
            |                                  |
            V                                  V
       |---------|                        |--------|
       |         |                        |        |
       | black   |                        | black  |
       | box     |                        | box    |
       |         |                        |        |
       |---------|                        |--------|
            |                                  |
            V                                  V
    reference output                         output
                                         who should match
                                       the reference output
```

Imagine you change some part of your program and want to know if it still works correctly. Just run your test framework with the input files and compare your output with the reference output. In case the two outputs are not identical, you will have to analyse the changes made in your program.

There are two possibilities now. Either your changes have undesired side effects and you have to implement the changes in some other way. This is the more common case. Or you have changed something in the logic of your program and you know that your new input is valid and correct. Then you have to change your test framework to include the new output as the new reference output.

Black box means that the test result do not tell anything about how jlint calculates its resulting output or whether the algorithms used are programmed correctly. To do this, one would use unit testing.

Unit testing is a kind of white box testing where you look at the internals of the box. You try to test single components for a specified behaviour. Unit testing could be implemented in a future version of the test framework.

A small test framework was added to `jlint` to allow testing the behaviour of the program. The test framework consists of a sample of class

files together with the output that `jlint` generates when run on these files. The sample class files are chosen to cover most of `jlint's` possible outcomes. To produce this output `jlint` version 3.0 was used and the output generated was stored in *.out files. These *.out files represent the reference output which should be matched by all future versions of `jlint` unless one changes something in the logic of `jlint`. Whenever there will be changes in the source code of `jlint`, the test framework should be used to make sure that the output of the new version is the same as the reference output. To run the test framework you need to go into the test/ directory, where you execute `./testall.sh`.
Script `testall.sh` itself calls `runtest.sh`, `showdiff.sh` and `showerror.sh`.

Script `runtest.sh` runs jlint on the sample java class files and writes its output into the *.log and its error messages into *.err.

Script `showdiff.sh` compares the *.out generated by `runtest.sh` with the stored *.log files and in case of a difference makes a diff x.out,log

Script `showerror.sh` looks for *.err file with size bigger than 0 and produces a warning and the name of the file if this happens. should not produce any output if nothing goes wrong. Only in the case that `jlint` crashes, it reports which files caused `jlint` to abort.

```
Test Framework (added in Version 3.0)
=====================================

File locations
--------------
class files to be tested        tests/test#/
log and reference output files  tests/log/

Tests:
------
4 tests as of Version 3.0
test1 test.java
test2 class showing finally bug
test3 sample from java/io/* (1.3)
test4 sample from java/io/* (1.4)

                                    in case of
                                    error:
#           default         run     run
#test 1     log/1.out       1.log   1.err
```

```
#test 2     log/2.out       2.log   2.err
#test 3     log/3.out       3.log   3.err
#test 4     log/4.out       4.log   4.err
...

Usage:
------

  To run test suite using valgrind,
try "./testall.sh --valgrind"

  For info on how to run tests,
try "./testall.sh --help"
```

3 Adding New Test Cases

Adding new test cases to the test framework is quite simple. A new directory e.g., test5 has to be created and the class files to be tested must be copied into that directory. In file `testall.sh` the variable `NROFTESTS` which is currently set to 4 has to be changed to reflect the new number of test cases. First of all a new reference file e.g., 5.out has to be created in the directory `tests/log/`. One has to be careful to use a fully functional (bug-free) version of `jlint` to produce a new reference *.out file. Once this is done `testall.sh` can be called and will run the tests including the newly added ones.

4 Found Errors, Bug Fixes

4.1 try–catch–finally constructs in Java 1.3 and Java 1.4

Before `java 1.4`, byte code verifiers of the java virtual machine had difficulties verifying the correctness of exception handlers with a complex control flow. These complex exception handlers were the result of a try–finally or a try–catch–finally construct.

Starting with `java 1.4` this bug in the virtual machine was worked around by changing the compiler. The generated byte code remains the same, but the number and the range of the exception handlers was changed in case there is a finally statement in the code.

Below you can see a java source file, followed by the corresponding byte code and the exception table. Differences between `java 1.3` and `java 1.4` will be pointed out.

```
class SC {
```

```
    void m(boolean b) {
        try {
            if (b) return;
        } finally {
            b = false;
        }
    }
}
```

Both java compilers produce exactly the same byte code in this case.
lines 0 to 7 handle the case that b == true
lines 8 to 11 handle the case that b == false
lines 14 to 19 handle the case that an exception occurs during the try block.
lines 20 to 23 are for the `finally` statement which has to be executed in any case.

```
void m(boolean arg1)
Code(max_stack = 1, max_locals = 4, code_length = 26)
0:    iload_1                    //put b on top of stack
1:    ifeq            #8         //if b != 0
4:    jsr             #20        //execute finally block
7:    return                     //exit
8:    jsr             #20        //execute finally block
11:   goto            #25        //goto exit statement
14:   astore_2                   //store exception
15:   jsr             #20        //execute finally block
18:   aload_2                    //load exception
19:   athrow                     //rethrow exception because
                                 //no catch statement
20:   astore_3
21:   iconst_0                   //put 0 (false) on stack
22:   istore_1                   //b = false
23:   ret             %3         //return to stmt after jsr
25:   return                     //exit
```

In this code, compiled by `javac 1.3`, there is only one exception handler which is valid for lines 0 to 14 exclusive. This means it catches exceptions occurring on lines 0 to and including 11. Whereas in the exception table below, compiled by `javac 1.4` there is also just one exception handler. But the range is split twice. Lines 7 and 11, the "return" and the "goto" instruction are excluded from the range. Instructions such as "return" and "goto" can be excluded from the range because they can never throw an exception.

```
Exception handler(s) =
>From    To      Handler Type
0        14      14      <Any exception>(0)
```

```
Exception handler(s) =
>From    To      Handler Type
0        7       14      <Any exception>(0)
8        11      14      <Any exception>(0)
14       18      14      <Any exception>(0)
```

In the next section it is shown how jlint 2.3 failed to correctly interpret this new kind of exception table and how this bug was fixed in jlint version 3.0.

4.2 Context Handling in `Jlint`

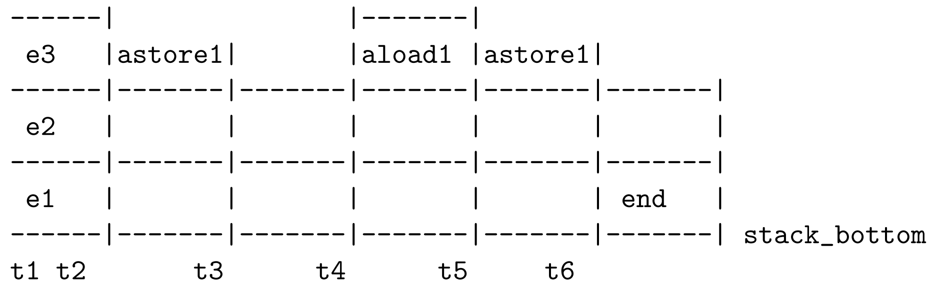
In its analysis of the class file jlint goes through the byte code calculating the range of possible values for each variable. For this a context data structure is used. In this context data structure the range of values a variable can have is saved. Contexts can be created, split and merged. Every byte code address has a linked list of contexts.

In the code fragment below which is from file `jlint.cc` for every entry in the exception table, such a context is inserted into the linked list at position "handler_pc" by calling "ctx_entry_point(&method->context[handler_pc]);".

```
while (--exception_table_length >= 0)
{
      int handler_pc = unpack2(fp+4);
      new ctx_entry_point(&method->context[handler_pc]);
      fp += 8;
}
```

After that the byte code instructions are analysed in jlint. Whenever `Jlint` starts the analysis of a new instruction, it goes through the linked list which corresponds to the instruction being analysed. The stack pointer is increased by one for each of the contexts found in the list. If for example there is an exception handler at position 14 which handles Exception $n > 1$. Then the linked list of position 14 has n entry point contexts amongst possibly other contexts. And so the stack pointer has been increased by n instead of only one. The problem here is that the stack pointer will only be reduced by one because there is only one exception handler in the byte code at a specific position, even if there is more than one range for the same exception handler. And so there is only one `astore` instruction where jlint will decrease

the stack pointer by one. Therefore after adding more than one exception context at the same byte code address the final assertion sp == stack_bottom fails and jlint returns with an error.



At time t2 the exception is stored and the stack pointer is decreased by one. But the stack pointer is still 2 positions too high and it will remain too high until the end, where the assertion assert(sp == stack_bottom) will fail.

4.3 Bug Related Errors

This bug caused jlint to exit abnormally under certain circumstances. Two different assertions could be violated because the simulated stack management didn't work properly any more.

The First kind of error occurred e.g., when analysing file `/java/io/BufferedReader.class`.

```
in file:    method_desc.cc
in method:  parse_code(constant **, const field_desc *)
 Assertion ‘sp == stack_bottom’ failed.
Aborted
```

The Second kind of error occurred e.g., when analysing file `/java/lang/FloatingDecimal.class`.

```
in file:    local_context.cc
in method:  transfer(...):
 Assertion ‘sp == come_from->stack_pointer’ failed.
Aborted
```

4.4 Minimal class file that reproduced the bug

Below is the listing of a minimal java class file which caused jlint to abort. This class file does not have a meaning, but still it is valid and gives a class file with a minimal number of byte code instructions.

```
class SC {
    void m() {
        try {
        } finally {
        }
    }
}
```

4.5 Bug fix

Here is the diff of the old and the new version of `jlint.cc`:

```
diff -u jlint_old.cc jlint.cc
--- jlint_old.cc        2003-04-26 15:29:23.000000000 +0200
+++ jlint.cc    2003-08-23 15:23:36.000000000 +0200
@@ -504,10 +504,45 @@
                sizeof(local_context*)*(code_length+1));

         int exception_table_length = unpack2(fp); fp += 2;
+
+       /* add new entry for each distinct "byte code address
+       ** of handle".
+       **
+       ** if an exception handler at byte code "pos" handles
+       ** exception of more than one byte code range, call
+       ** "new ctx_entry_point(&method->context[pos]);" only
+       ** once! Because otherwise the stack gets out of
+       ** control.
+       **
+       ** in the following example there are two different
+       ** handle addresses 16 and 25. and for each of them
+       **"new ctx_entry_point(&method->context[handler_pc]);"
+       ** is called exactly once. Therefore the program calls
+       ** new ctx_entry_point(&method->context[16]);
+       ** new ctx_entry_point(&method->context[25]);
+       ***************************************************
+       ** Example Exception Table:                       **
+       ** -------------------------------------          **
+       **                                               **
+       **                      byte code address          **
+       ** from         to        of handle               **
+       **  2           10           16                   **
+       ** 12           14           16                   **
```

```
+       ** 20           23           25                      **
+       ****************************************************
+       **
+       ** it is expected that the byte code addresses of the
+       ** handles are ordered. If this would not be the case,
+       ** a simple comparison of handler_pc and
+       ** old_handler_pc would not be sufficient!
+       */
+
+       int old_handler_pc = -1;
+
         while (--exception_table_length >= 0) {
           int handler_pc = unpack2(fp+4);
-          new ctx_entry_point(&method->context[handler_pc]);
-          fp += 8;
+         if ( handler_pc != old_handler_pc) {
+           new ctx_entry_point(&method->context[handler_pc]);
+         }
+         fp += 8;
+         old_handler_pc = handler_pc;
         }

         int method_attr_count = unpack2(fp); fp += 2;
```

5 Changes to the Build Process

A new and automated build process was being started to work on during this Semester project. Before, there was no ./configure. The `Makefile` had to be changed manually if one wanted to set some architecture specific flags or if one wanted a debugging build or a build for a different target machine. The dependencies for compilation are automatically generated using a perl script: `mkmf.pl`. A very basic configure script was added which calculates the options and sets environment variables depending on the Operating System. This configure script generates the `Makefile`. More precisely it takes a standard `Makefile.in` and replaces the unspecified options by values it calculates.

The new target test_dist was added to the `Makefile`. "make test_dist" builds a tar.gz of the sources including the test directory with all the test files. It is intended to be used by future developers of `jlint` to be able to check for errors.

6 Results and Conclusion

What has been done and what still needs to be done:
The sources can be compiled on Intel IA32 Architecture using Linux as OS and GCC 3.X as compiler. The finally bug was fixed. The file `method_desc.cc` got a better documentation. Two patches got merged. A basic configure script has been written (only works for linux on IA32 and not even here it is guaranteed to work). A test framework was added which should make it easier to check for errors in the future.

The configure script should be improved and support for other architectures as well as other operating systems should be added. The cvs repository should be moved to the sourceforge account. Support for 64-bit architectures needs to be implemented as well. The valgrind tests in the test framework don't really work yet. The problem is how to run valgrind with shell scripts. Some more things which one could add to improve jlint and to fix some open bugs can be found in the files BUGS and TODO which come with `jlint`.