,

Jlint manual

Java program checker

Konstantin Knizhnik, Cyrille Artho

Table of Contents

jlint

This document described Jlint, a Java program checker that will check your Java code and find bugs, inconsistencies and synchronization problems by doing data flow analysis and building a lock graph.

1 Introduction

Jlint will check your Java code and find bugs, inconsistencies and synchronization problems by doing data flow analysis and building lock graph.

Jlint consists of two separate programs performing syntax and semantic verification. As far as Java mostly inherits C/C++ syntax and so inherits most of the problems caused by C syntax, the idea was to create common syntax verifier for all C-family languages: C, C++, Objective C and Java. This program was named *AntiC*, because it fixes problems with C grammar, which can cause dangerous programmer's bugs, undetected by compiler. By using hand-written scanner and simple top-down parser, AntiC is able to detect such bugs as suspicious use of operators priorities, absence of break in switch code, wrong assumption about constructions bodies...

Semantic verifier Jlint extracts information from Java class files. As far as Java class file has very well specified and simple format, it greatly simplifies Jlint in comparison with source level verifiers, because development of Java grammar parser is not a simple task (even through Java grammar is simpler and less ambiguous than C++ grammar). Also dealing only with class files, protect Jlint from further Java extensions (format of virtual byte instructions is more conservative). By using debugging information Jlint can associate reported messages with Java sources.

Jlint performs local and global data flow analyses, calculating possible values of local variables and catching redundant and suspicious calculations. By performing global method invocation analysis, Jlint is able to detect invocation of method with possible "null" value of formal parameter and using of this parameter in method without check for "null". Jlint also builds lock dependency graph for classes dependencies and uses this graph to detect situations, which can cause *deadlock* during multi-threaded program execution. Except deadlocks, Jlint is able to detect possible *race condition* problem, when different threads can concurrently access the same variables. Certainly Jlint can't catch all synchronization problems, but at least it can do something, which can save you a lot of time, because synchronization bugs are the most dangerous bugs: non-deterministic, and not always reproducible. Unfortunately Java compiler can't help you with detecting synchronization bugs, may be Jlint can...

Jlint uses smart approach to message reporting. All messages are grouped in categories, and it is possible to enable or disable reporting messages of specific category as well as concrete messages. Jlint can remember reported messages and do not report them once again when you run Jlint second time. This feature is implemented by means of history file. If you specify *-history* option, then before reporting a message, Jlint searches in this file if such message was already reported in the past. If so, then no message is reported and programmer will not have to spend time parsing the same messages several times. If message was not found in history file, it is reported and appended to history file to eliminate reporting of this message in future. Some messages refer to class/method name and are position independent, while some messages are reported for specific statement in method's code. Messages of second type will not be repeatedly reported only if method's source is not changed.

2 Bugs detected by AntiC

Input of AntiC should be valid C/C++ or Java program with no syntax errors. If there are some syntax errors in the program, AntiC can detect some of them and produce error message, but it doesn't try to perform full syntax checking and can't recover after some errors. So in this chapter we discuss only the messages produced by AntiC for program without syntax errors.

2.1 Bugs in tokens

2.1.1 Octal digit expected

Sequence of digits in string or character constant preceded by '\\' character contains non-octal digit:

```
printf("\128");
```

2.1.2 May be more than three octal digits are specified

Sequence of digits in string or character constant preceded by '\\' character contains more than three digits:

```
printf("\1234");
```

2.1.3 May be more than four hex digits are

specified for character constant

String constant contains escape sequence for Unicode character, followed by character, which can be treated as hexadecimal digit:

```
System.out.println("\uABCDE:");
```

2.1.4 May be incorrect escape sequence

Non-standard escape sequence is used in character or string constant:

```
printf("\x");
```

2.1.5 Trigraph sequence inside string

Some C/C++ compilers still support trigraph sequences of ANSI C and replace the following sequences of characters ("??=", "??/", "??'", "??(", "??)", "??!", "??") with the characters ("#", "\", "^", "[", "]", "|", "", "") respectively. This feature may cause unexpected transformation of string constants:

```
char* p = "???=undefined";
```

2.1.6 Multi-byte character constants are not portable

Multi-byte character constants are possible in C, but makes program non-portable.

```
char ch = 'ab';
```

2.1.7 May be 'l' is used instead of '1' at the end of integer constant

It is difficult to distinct lower case letter 'l' and digit '1'. As far as letter 'l' can be used as long modifier at the end of integer constant, it can be mixed with digit. It is better to use upper-case 'L':

```
long l = 0x1111111;
```

2.2 Operator priorities

2.2.1 May be wrong assumption about operators precedence

Several operators with non-intuitive clear precedence are used without explicit grouping by parentheses. Sometimes programmer's assumption about operators priorities is not true, and in any case enclosing such operations in parentheses can only increase readability of program. Below is list of some suspicious combinations of operators:

```
x & y == z
x && y & z
x || y = z
```

2.2.2 May be wrong assumption about logical operators precedence

Priority of logical AND operator is higher than priority of logical OR operator. So AND expression will be evaluated before OR expression even if OR precedes AND:

```
x || y && z
```

2.2.3 May be wrong assumption about shift operator priority

Priority of shift is smaller than of arithmetic operators but less than of bit manipulation operators. It can cause wrong assumption about operands grouping:

```
x>>y - 1
x >> y&7
```

2.2.4 May be '=' used instead of '=='

Almost all C programmer did this bug, at least once in their life. It very easy to type '=' instead of '==' and not all C compilers can detect this situation. Moreover this bug is inherited by Java: the only restriction is that types of operands should be boolean:

```
if (x = y)
```

2.2.5 May be skipped parentheses around assign operator

Assign operators have one of the smallest priorities. So if you want to test result of assignment operation, you should enclose it in parentheses:

```
if (x>>=1 != 0)
```

2.2.6 May be wrong assumption about bit operation priority

Bit manipulation operators have smaller priority than compare operators. If you, for example, extracting bits using bit AND operator, do not forget to enclose it with parentheses, otherwise result of the expression will be far from your expectation:

```
if (x == y & 1)
```

2.3 Statement body

Almost all C statements can contain as its sub-part either single statement or block of statements (enclosed by braces). Unnoticed semicolon or wrong alignment can confuse programmer about real statement's body. And compiler can't produce any warnings, because it deals with stream of tokens, without information about code alignment.

2.3.1 May be wrong assumption about loop body

This message is produced if loop body is not enclosed in braces and indentation of the statement following the loop is bigger than of loop statement (i.e. it is shifted right):

```
while (x != 0)
    x >>= 1;
    n += 1;
return x;
```

2.3.2 May be wrong assumption about IF body

This message is produced if IF body is not enclosed in braces and indentation of the statement following the IF construction is bigger than of IF statement itself (i.e. it is shifted right) or IF body is empty statement (';'):

```
if (x > y);

    int tmp = x;
    x = y;
    y = tmp;


if (x != 0)
    x = -x; sign = -1;
sqr = x*x;
```

2.3.3 May be wrong assumption about ELSE branch association

If there are no braces, then ELSE branch belongs to most inner IF. Sometimes programmers forget about it:

```
if (rc != 0)
    if (perr) *perr = rc;
else return Ok;
```

2.3.4 Suspicious SWITCH without body

Switch statement body is not a block. With great probability it signals about some error in program:

```
switch(j)
  case 1:
    ...
  case 2:
    switch(ch);
```

```
case 'a':
case 'b':
  ...
```

2.3.5 Suspicious CASE/DEFAULT

Case is found in block not belonging to switch operator. Situations, where such possibility can be used are very rare:

```
switch (n & 3)
    do
        default:
            *dst++ = 0;
        case 3:
            *dst++ = *drc++;
        case 2:
            *dst++ = *drc++;
        case 1:
            *dst++ = *drc++;
     while ((n -= 4) > 0;
```

2.3.6 Possible miss of BREAK before CASE/DEFAULT

AntiC performs some kind of control flow analysis to detect situations, where control can be passed from one case branch to another (if programmer forget about BREAK statement). Sometimes it is necessary to merge several branches. AntiC doesn't produce this message in following cases:

1. Several cases point to the same statement:

```
case '+':
case '-':
  sign = 1;
  break;
```

2. Special `nobreak` macro is defined and used in switch statement:

```
#define nobreak
...
switch (cop)
  case sub:
    sp[-1] = -sp[1];
    nobreak;
  case add:
    sp[-2] += sp[-1];
    break;
    ...
```

3. Comment containing words "break" or "fall" is placed before the case:

```
switch (x)
  case do_some_extra_work:
    ...
    // fall thru
  case do_something:
    ...
```

In all other cases message is produced when control can be passed from one switch branch to another:

```
switch (action)
  case op_remove:
    do_remove();
  case op_insert:
    do_insert();
  case op_edit:
    do_edit();
```

3 Bugs detected by Jlint

There are three main groups of messages produced by Jlint: *synchronization, inheritance* and *data flow*. These groups are distinguished by kind of analysis which is used to detect problems, reported in this messages. Each group is in turn divided into several categories, which contains one or more messages. Such scheme of message classification is used to support fine-grained selection of reported messages.

Because only categories of message can be disabled, but not separate messages, a short shell script is supplied that will suppress certain warnings that are less important. It should be noted that it will ignore any race conditions for variables. This is because Jlint does not have any notion of "shared reading", so it usually produces too many warnings about such data races to be useful. For some projects, using Jlint's output unfiltered can still be useful.

The shell script is as follows:

```
#!/bin/bash
find . -name '*.class' | xargs jlint -not_overridden \
-redundant -weak_cmp -bounds -zero_operand -string_cmp -shadow_local | \
grep -v 'Field .class\$' | grep -v 'can be .*ed from different threads' | \
grep -v 'Method.*Runnable.*synch'
```

It is probably more easier to run Jlint using those filters, with

```
cd <dir_with_class_files>
jlint.sh
```

3.1 Synchronization

Parallel execution of several threads of control requires some synchronization mechanism to avoid access conflicts to shared data. Java approach to synchronization is based on using object monitors, controlled by `synchronized` language construction. Monitor is always associated with object and prevents concurrent access to the object by using mutual exclusion strategy. Java also supports facilities for waiting and notification of some condition.

Unfortunately, providing these synchronization primitives, Java compiler and virtual machine are not able to detect or prevent synchronization problems. Synchronization bugs are the most difficult bugs, because of non-deterministic behaviour of multi-threaded program. There are two main sources of synchronization problems: deadlocks and race conditions.

Situation in which one or more threads mutually lock each other is called deadlock. Usually the reason of deadlock is inconsistent order of resource locking by different threads. In Java case resources are object monitors and deadlock can be caused by some sequence of method invocations. Let's look at the following example of multi-threaded database server:

```
class DatabaseServer
    public TransactionManager transMgr;
    public ClassManager       classMgr;
    ...

class TransactionManager
    protected DatabaseServer server;
```

```
        public synchronized void commitTransaction(ObjectDesc[] t_objects)
            ...
            for (int i = 0; i < t_objects.length; i++)
                ClassDesc desc = server.classMgr.getClassInfo(t_objects[i]);
                ...

            ...

        ...

    class ClassManager
        protected DatabaseServer server;

        public synchronized ClassDesc getClassInfo(ObjectDesc object)
            ...

        public synchronized void addClass(ClassDesc desc)
            ObjectDesc t_objects;
            ...
            // Organized transaction to insert new class in database
            server.transMgr.commit_transaction(t_objects);

    ;
```

If database server has one thread for each client and one client is committing transaction while another client adds new class to database, then deadlock can arise. Consider the following sequence:

1. Client A invokes method `TransactionManager.commitTransaction()`. While execution of this method monitor of TransactionManager object is locked.
2. Client B invokes method `ClassManager.addClass()` and locks monitor of ClassManager object.
3. Method `TransactionManager.commitTransaction()` tries to invoke method `ClassManager.getClassInfo()` but has to wait because this object is locked by another thread.
4. Method `ClassManager.addClass()` tries to invoke method `TransactionManager.commitTransaction()` but has to wait because this object is locked by another thread.

So we have deadlock and database server is halted and can't serve any client. The reason of this deadlock is loop in locking graph. Let's explain it less formally. We will construct oriented graph G of monitor lock relations. As far as locked resource are objects, so vertexes of this graph should be objects. But this analysis can't be done statically, because set of all object instances is not known at compile time. So the only kind of analysis, which Jlint is able to perform, is analysis of inter-class dependencies. So the vertexes of graph G will be classes. More precisely, each class C is represented by two vertexes: vertex C for class itself and vertex C' for metaclass. First kind of vertexes are used for dependencies caused by instance methods invocation, and second - by static methods. We will add edge (A,B)

with mark "foo" to the graph if some synchronized method `foo()` of class B, can be invoked directly or indirectly from some synchronized method of class A for object other than `this`. For example for the following classes:

```
class A
    public synchronized void f1(B b)
        b.g1();
        f1();
        f2();

    public void f2(B b)
        b.g2();

    public static synchronized void f3()
        B.g3();


class B
    public static A ap;
    public static B bp;
    public synchronized void g1()
        bp.g1();

    public synchronized void g2()
        ap.f1();

    public static synchronized void g3()
        g3();
```

will add the following edges:

```
      g1
A  --------> B,  because of invocation of b.g1() from A.f1()

      g2
A  --------> B,  because of following call sequence: A.f1 -> A.f2 -> B.g2

      g3
A' --------> B', because of invocation of b.g3() from A.f3()

      g1
B  --------> B,  loop edge because of recursive call for non-this object in B.g1().

      f1
B  --------> A,  because of invocation of ap.f1() from B.g2()
```

Deadlock is possible only if there is loop in graph G. This condition is necessary, but not enough (presence of loop in graph G doesn't mean that program is not correct and deadlock

can happen during it's execution). So using this criterion Jlint can produce messages about deadlock probability in case where deadlock is not possible.

As far as task of finding all loops in the graph belongs to the NP class, no efficient algorithm for reporting all such loops exists at this moment. To do it work best and fast, Jlint uses restriction for number of loops, which pass through some graph vertex.

There is another source of deadlock - execution of `wait()` method. This method unlocks monitor of current object and waits until some other thread notify it. Both methods `wait()` and `notify()` should be called with monitor locked. When thread is awaken from wait state, it tries to re-establish monitor lock and only after it can continue execution. The problem with `wait()` is that only one monitor is unlocked. If method executing `wait()` was invoked from synchronized method of some other object O, monitor of this object O will not be released by `wait`. If thread, which should notify sleeping thread, needs to invoke some synchronized method of object O, we will have deadlock: one thread is sleeping and thread, which can awoke it, waits until monitor will be unlocked. Jlint is able to detect situations when `wait()` method is called and more than one monitors are locked.

But deadlock is not the only synchronization problem. Race condition or concurrent access to the same data is more serious problem. Let's look at the following class:

```
class Account
    protected int balance;

    public boolean get(int sum)
        if (sum > balance)
            balance -= sum;
            return true;

        return false;
```

What will happen if several threads are trying to get money from the same account? For example account balance is $100. First thread tries to get $100 from the account - check is ok. Then, before first thread can update account balance, second thread tries to perform the same operation. Check is ok again! This situation is called *race condition*, because result depends on "speed" of threads execution.

How can Jlint detect such situations? First of all Jlint builds closure of all methods, which can be executed concurrently. The obvious candidates are synchronized methods and method `run` of classes implemented `Runnable` protocol or inherited from `Thread` class. Then all other methods, which can be invoked from these methods, are marked as concurrent. This process repeats until no more method can be added to concurrent closure. Jlint produces message about non-synchronized access only if all of the following conditions are true:

1. Method accessing field is marked as concurrent.
2. Field is not declared as `volatile` or `final`.
3. Field doesn't belong to `this` object of the method.
4. It is not a field of just created object, which is accessed through local variable.

5. Field can be accessed from methods of different classes.

It is necessary to explain last two items. When object is created and initialized, usually only one thread can access this object through its local variables. So synchronization is not needed in this case. The explanation of item 5 is that not all objects, which are accessed by concurrent threads, need to be synchronized (and can't be declared as synchronized in some cases to avoid deadlocks). For example consider implementation of database set:

```
class SetMember
    public SetMember next;
    public SetMember prev;

class SetOwner
    protected SetMember first;
    protected Setmember last;

    public synchronized void add_first(SetMember mbr)
        if (first == null)
            first = last = mbr;
            mbr.next = mbr.prev = null;
         else
            mbr.next = first;
            mbr.prev = null;
            first.prev = mbr;
            first = mbr;


    public synchronized void add_last(SetMember mbr) ...
    public synchronized void remove(SetMember mbr) ...
;
```

In this example `next` and `prev` components of class `SetMember` can be accessed only from synchronized methods of `SetOwner` class, so no access conflict is possible. Rule 5 was included to avoid reporting of messages in situations like this.

Rules for detecting synchronization conflicts by Jlint are not finally defined, some of them can be refused or replaced, new candidates can be added. The main idea is to detect as much suspicious places as possible, while not producing confusing messages for correct code.

3.1.1 Loop *id*: invocation of synchronized method *name* can cause deadlock

Message category: **deadlock**
Message code: **sync_loop**

Loop in class graph G See Section 3.1 [Synchronization], page 8 is detected. One such message is produced for each edge of the loop. All loops are assigned unique identifier, so it is possible to distinguish messages for edges of one loop from another.

3.1.2 Loop *LoopId/PathId*: invocation of method *name* forms the loop in class dependency graph

Message category: **deadlock**
Message code: **loop**

Reported invocation is used in call sequence from synchronized method of class A to synchronized method `foo()` of class B, so that edge (A,B) is in class graph G (See Section 3.1 [Synchronization], page 8). If method `foo()` is invoked directly, then only previous message (sync_loop) is reported. But if call sequence includes some other invocations (except invocation of `foo()`), then this message is produced for each element of call sequence. If several call paths exist for classes A, B and method `foo()`, then all of them (but not more than specified by `MaxShownPaths` parameter) are printed. *PathId* identifier is used to group messages for each path.

3.1.3 Lock *a* is requested while holding lock *b*, with other thread holding *a* and requesting lock *b*

Message category: **deadlock**
Message code: **lock**

This is one of the extensions for version 2: checking `synchronized` blocks. If, for one class, the locking scheme is such that it could lead to a cycle in the locking graph, this message is shown.

```
public void foo()
  synchronized (a)
    synchronized (b)



public void bar()
  synchronized (b)
    synchronized (a)
```

In this example, `a` and `b` are two objects that are used as locks and are shared between threads. If one thread call `foo` while another one calls `bar` simultaneously, a deadlock occurs. Jlint does not check whether `a` and `b` are actually used by several threads. However, if this were not the case, there is no point of using synchronizations on these variables.

3.1.4 Method wait() can be invoked with monitor of other object locked

Message category: **deadlock**
Message code: **wait**

At the moment of `wait()` method invocations, more than one monitor objects are locked by the thread. As far as wait unlocks only one monitor, it can be a reason of deadlock. Successive messages of type **wait_path** specify call sequence, which leads to this invocation. Monitors can be locked by invocation of a synchronized method or by explicit synchronized construction. Jlint handle both of the cases.

The extended Jlint now checks *which* locks are actually owned before issuing an error message. This error message now spans two lines, with the second line saying which locks are owned at that point. (Jlint will still count this as only one message when printing the total message count.) This should greatly facilitate debugging.

3.1.5 Call sequence to method *name* can cause deadlock in wait()

Message category: **deadlock**
Message code: **wait_path**

By the sequence of such messages Jlint informs about possible invocation chain, which locks at least two object monitors and is terminated by method calling `wait()`. As far as `wait()` unlocks only one monitor and suspend thread, this can cause deadlock.

3.1.6 Synchronized method *name* is overridden by non-synchronized method of derived class *name*

Message category: **race_condition**
Message code: **nosync**

Method is declared as synchronized in base class, but is overridden in derived class by non-synchronized method. It is not a bug, but suspicious place, because if base method is declared as synchronized, then it is expected that this method can be called from concurrent threads and access some critical data. Usually the same is true for derived method, so disappearance of synchronized modifier looks suspiciously.

3.1.7 Method *name* can be called from different threads and is not synchronized

Message category: **race_condition**
Message code: **concurrent_call**

Non-synchronized method is invoked from method marked as concurrent for object other than `this` (for instance methods) or for class, which is not base class of caller method class (for static methods). This message is reported only if invocation is not enclosed in synchronized construction and this method also can be invoked from methods of other classes.

3.1.8 Field *name* of class

name can be accessed from different threads and is not volatile

Message category: **race_condition**
Message code: **concurrent_access**

Field is accessed from method marked as concurrent. This message is produced only if:

1. Field belongs to the object other than `this` (for instance methods) or to classes which are not base for class of static method.
2. Field is not component of object previously created by `new` and assigned to local variable.
3. Field is not marked as volatile or final.
4. Field can be accessed from methods of different classes.

3.1.9 Method *name* implementing 'Runnable' interface is not synchronized

Message category: **race_condition**
Message code: **run_nosync**

Method `run()` of class implementing `Runnable` interface is not declared as synchronized. As far as different threads can be started for the same object implementing `Runnable` interface, method `run` can be executed concurrently and is first candidate for synchronization.

3.1.10 Value of lock *name* is changed outside synchronization or constructor

Message category: **deadlock**
Message code: **loop_assign**

```
class Foo
  Object a = new Object();

  public void bar()
    a = new Object();
    synchronized (a)
```

The initialization of *a* (in the declaration, which will be moved into the constructor) is OK; however, changing the value outside any synchronization will make *a* useless as a locking variable. Therefore, Jlint will issue a warning for the assignment `a = new Object();` in `bar`.

3.1.11 Value of lock *name* is changed while (potentially) owning it

Message category: **deadlock**
Message code: **loop_assign2**

```
class Quux
  Object a = new Object();

  public void foo()
    synchronized (a)
      bar();



  public void bar()
    a = new Object();
    /* do something */
```

In this example, the current thread still holds a lock on *a* when it re-initialized that variable (in method `bar`). This means that if another thread tried to obtain a lock on the new *a*, it can now proceed to do so, because the new value of *a* points to a different instance, which makes a synchronization on *a* ineffective. Probably this was not expected

by the programmer, and this could lead to a potential race condition. The solution to this problem is to include another guard (in this case, `synchronized(this)`).

3.1.12 Method *name*`.wait()` is called without synchronizing on *name*

Message category: **wait_nosync**
Message code: **wait_nosync**

Method `wait()` or `notify()` is invoked from method, which is not declared as synchronized. It is not surely a bug, because monitor can be locked from another method, which directly or indirectly invokes current method.

The improved Jlint (version 2) can also check `wait` calls to any object, and it will not report an error as long as the lock on *name* was obtained within the method that is currently being checked. This greatly reduces the amount of spurious warnings in that category.

3.2 Inheritance

This group contains messages, which are caused by problems with class inheritance: such as mismatch of methods profiles, components shadowing... As far as Jlint deals with Java class file and there is no information about line number in source file of class, field or method definition, Jlint can't show proper place in source file where class, field or method, which cause the problem, is located. In case of methods, Jlint points to the line corresponds to the first instruction of the method. And for classes and fields, Jlint always refers in message to the first line in source file. Jlint assign successive number (starting from 1) for all such message reported sequentially, because Emacs skips all messages, reported for the same line, when you go to next message.

3.2.1 Method *name* is not overridden by method with the same name of derived class *name*

Message category: **not_overridden**
Message code: **not_overridden**

Derived class contains the method with the same name as in base class, but profiles of these methods do not match. More precisely: message is reported when for some method of class A, exists method with the same name in derived class B, but there is no method with the same name in class B, which is compatible with definition of the method in class A (with the same number and types of parameters). Programmer writing this code may erroneously expect that method in derived class overrides method in base class and that virtual call of method of base class for object of derived class will cause execution method of the derived class.

3.2.2 Component *name* in class *name* shadows one in base class *name*

Message category: **field_redefined**
Message code: **field_redefined**

Field in derived class has the same name as field of some of base classes. It can cause some problems because this two fields points to different locations and methods of base class will access one field, while methods of derived class (and classes derived from it) will

access another field. Sometimes it is what programmer expected, but in any case it will not improve readability of program.

3.2.3 Local variable *name* shadows component of class *name*

Message category: **shadow_local**
Message code: **shadow_local**

Local variable of method shadows class component with the same name. As far as it is common practice in constructors to use formal parameters with the same name as class components, Jlint detects situations, when class field is explicitly accessed by using `this` reference and doesn't report this message in this case:

```
class A
    public int a;
    public void f(int a)
        this.a = a; // no message

    public int g(int a)
        return a; // message "shadow_local" will be reported
```

3.2.4 Method `finalize()` doesn't call `super.finalize()`

Message category: **super_finalize**
Message code: **super_finalize**

As it is mentioned in book "The Java Programming Language" by Ken Arnold and James Gosling, calling of `super.finalize()` from `finalize()` is good practice of programming, even if base class doesn't define `finalize()` method. This makes class implementations less dependent from each other.

3.3 Data flow

Jlint performs data flow analysis of Java byte code, calculating possible ranges of values of expressions and local variables. For integer types, Jlint calculates minimal and maximal value of expression and mask of possibly set bits. For object variables attribute `null/not_null` is calculated, selecting variables which value can be `null`. When value of expression is assigned to variable, these characteristics are copied to correspondent variable descriptor. Jlint handles control transfer instruction in special way: saving, modifying, merging or restoring context depending on type of instruction. Context in this consists of local variables states (minimal, maximal values and mask) and state of top of the stack (for handling ?: instruction). Initially all local integer variable are considered to have minimum and maximum properties equal to the range of correspondent type, and mask indicating that all bits in this range can be set. Object variables attribute initially is set to `not_null`. The same characteristics are always used for class components, because Jlint is not able to perform full data flow analysis (except checking for passing null value to formal parameter of methods). Table below summarizes actions performed by Jlint for handling control transfer instruction:

Instruction type	Correspondent Java construction	Action
Forward conditional jump	IF statement	Save current context. Modify current context in assumption that condition is false (no jump). Modify saved context in assumption that condition is true (jump takes place)
Forward unconditional jump	Start of loop, jump around ELSE branch of IF	Save current context
Backward conditional jump	Loop statement condition	Modify context in assumption that condition is false (no jump)
Backward unconditional jump	Infinite loop	Do nothing
Label of forward jump	End of IF body or SWITCH case	If previous instruction is no-pass instruction (return, unconditional jump, throw exception) then restore saved context, otherwise merge current context with saved context (set minimum property of integer variable to minimum of this property value in current and saved contexts, maximum - to maximum of values in two contexts, and mask as join of masks in two context; for object variable - mark it as "may contain null" if it is marked so in one of contexts). If label corresponds to switch statement case, and switch expression is single local variable, then update properties of this variable by setting its minimum and maximum values and mask to value of case selector.

Label of backward jump	Start of loop body	Reset properties of all variables modified between this label and backward jump instructions. Reset for integer variables means setting minimum property to minimum value of correspondent type, ... Reset for object variable clears mark "may contain null".

3.3.1 Method *name* can be invoked with NULL as *number* parameter and this parameter is used without check for null

Message category: **null_reference**
Message code: **null_param**

Formal parameter is used in the method without check for null (component of object is accessed or method of this object is invoked), while this method can be invoked with null as the value of this parameter (detected by global data flow analysis). Example:

```
class Node
    protected Node next;
    protected Node prev;
    public void link(Node after)
        next = after.next; // Value of 'after' parameter can be null
        prev = after;
        after.next = next.prev = this;


class Container
    public void insert(String key)
        Node after = find(key);
        if (after == null)
            add(key);

        Node n = new Node(key);
        n.link(after); // after can be null
```

3.3.2 Value of referenced variable *name* may be NULL

Message category: **null_reference**
Message code: **null_var**

Variable is used in the method without check for null. Jlint detects that referenced variable was previously assigned `null` value or was found to be `null` in one of control paths in the method.

Jlint can produce this message in some situations, when value of variable can not actually be null:

```
public int[] create1nVector(int n)
    int[] v = null;
    if (n > 0)
        v = new int[n];

    for (int i = 0; i &lt; n; i++)
        v[i] = i+1; // message will be reported

    return v;
```

3.3.3 NULL reference can be used

Message category: **null_reference**
Message code: **null_ptr**

Constant `null` is used as left operand of '.' operation:

```
public void printMessage(String msg)
    (msg != null ? new Message(msg) : null).Print();
```

3.3.4 Zero operand for operation

Message category: **zero_operand**
Message code: **zero_operand**

One of operands of binary operation is zero. This message can be produced for sequence of code like this:

```
int x = 0;
x += y;
```

3.3.5 Result of operation is always 0

Message category: **zero_result**
Message code: **zero_result**

Jlint detects that for given operands, operation always produces zero result. This can be caused by overflow for arithmetic operations or by shifting all significant bits in shift operations or clearing all bits by bit AND operation.

3.3.6 Shift with count *relation* than *integer*

Message category: **domain**
Message code: **shift_count**

This message is reported when minimal value of shift count operand exceeds 31 for int type and 63 for long type or maximal value of shift count operand is less than 0:

```
if (x > 32)
    y >>= x; // Shift right with count greater than 32
```

3.3.7 Shift count range [*min*,*max*] is out of domain

Message category: **domain**
Message code: **shift_count**

Range of shift count operand is not within [0,31] for int type or [0,63] for long type. Jlint doesn't produce this message when distance between maximum and minimum values of shift count is greater than 255. So this message will not be reported if shift count is just variable of integer type:

```
public int foo(int x, int y)
    x >>= y; // no message
    x >>= 32 - (y & 31); // range of count is [1,32]
```

3.3.8 Range of expression value has no intersection with *target* type domain

Message category: **domain**
Message code: **conversion**

Converted value is out of range of target type. This message can be reported not only for explicit conversions, but also for implicit conversions generated by compiler:

```
int x = 100000;
short s = x; // will cause this message
```

3.3.9 Data can be lost as a result of truncation to *type*

Message category: **truncation**
Message code: **truncation**

This message is reported when significant bits can be lost as a result of conversion from large integer type to smaller. Such conversions are always explicitly specified by programmer, so Jlint tries to reduce number of reported messages caused by data truncation. Example below shows when Jlint produces this message and when not:

```
public void foo(int x, long y)
    short s = (short)x; // no message
    char  c = (char)x;  // no message
    byte  b = (byte)y;  // no message
    b = (byte)(x & 0xff); // no message
    b = (byte)c; // no message
    c = (x & 0xffff); // no message
    x = (int)(y >>> 32); // no message


    b = (byte)(x >> 24);     // truncation
    s = (int)(x & 0xffff00); // truncation
    x = (int)(y >>> 1);      // truncation
    s = (short)c;            // truncation
```

3.3.10 May be type cast is not correctly applied

Message category: **overflow**
Message code: **overflow**

Result of operation, which has good chance to cause overflow (multiplication, left shift), is converted to long. As far as operation is performed with `int` operands, overflow can happen before conversion. Overflow can be avoided by conversion of one of operation operands to long, so operation will be performed with `long` operands. This message is produced not only for explicit type conversion done by programmer, but also for implicit type conversions performed by compiler:

```
public long multiply(int a, int b)
    return a*b; // operands are multiplied as integers
                // and then result will be converted to long
```

3.3.11 Comparison always produces the same result

Message category: **redundant**
Message code: **same_result**

Using information about possible ranges of operands values, Jlint can make a conclusion, that logical expression is always evaluated to the same value (`true` or `false`):

```
public void foo(int x)
    if (x > 0)
        ...
        if (x == 0) // always false
```

3.3.12 Compared operands can be equal only when both of them are 0

Message category: **redundant**
Message code: **disjoint_mask**

By comparing operands masks, Jlint makes a conclusion that operands of `==` or `!=` operations can be equal only when both of them are zero:

```
public boolean foo(int x, int y)
    return ((x & 1) == y*2); // will be true only for x=y=0
```

3.3.13 Reminder always equal to the first operand

Message category: **redundant**
Message code: **redundant**

This message is produced for `%` operation when right operand is either greater either less than zero, and absolute value of left operand is less than absolute value of right operand. In this case `x % y == x` or `x % y == -x`.

3.3.14 Comparison of short with char

Message category: **short_char_cmp**
Message code: **short_char_cmp**

Comparison of `short` operand with `char` operand. As far as `char` type is unsigned, and is converted to `int` by filling high half of the word with 0, and `short` type is signed and is converted to `int` using sign extension, then symbols in range `0x8000...0xFFFF` will not be considered equal in such comparison:

```
boolean cmp()
   short s = (short)0xabcd;
   char c = (char)s;
   return (c == s); // false
```

3.3.15 Compare strings as object references

Message category: **string_cmp**
Message code: **string_cmp**

String operands are compared by `==` or `!=` operator. As far as `==` returns `true` only if operands point to the same object, so it can return false for two strings with same contents. The following function will return `false` in JDK1.1.5:

```
public boolean bug()
    return Integer.toString(1) == Integer.toString(1);
```

3.3.16 Inequality comparison can be replaced with equality comparison

Message category: **weak_cmp**
Message code: **weak_cmp**

This message is produced in situations when ranges of compared operands intersect only in one point. So inequality comparison can be replaced with equality comparison. Such message can be caused by error in program, when programmer has wrong assumption about ranges of compared operands. But even if this inequality comparison is correct, replacing it with equality comparison can make code more clear:

```
public void foo(char c, int i)
    if (c &lt;= 0)  // is it a bug ?
        if ((i & 1) > 0)  // can be replaced with (i & 1) != 0
            ...
```

3.3.17 Switch case constant *integer* can't be produced by switch expression

Message category: **incomp_case**
Message code: **incomp_case**

Constant in switch case is out of range of switch expression or has incompatible bit mask with switch expression:

```
public void select(char ch, int i)
    switch (ch)
      case 1:
      case 2:
      case 3:
        ...
      case 256: // constant is out of range of switch expression

    switch (i & ~1)
      case 0:
      case 0xabcde:
        ...
      case 1: // switch expression is always even
```

4 Command line options

Both programs (AntiC and Jlint) accept list of files separated by spaces in command line. Wildcards are permitted. But unlike Unix, where wildcards are substituted by shell, in Windows wildcards are handled by program itself and wildcards only in file names (not in path directories) are allowed.

4.1 AntiC command line options

AntiC supports only one command option: `"-java"`. By default it consider input files as C/C++ source. There are very few differences (from AntiC point of view) between Java and C++. The differences are mostly with set of tokens and Unicode character constants.

4.2 Jlint command line options

Jlint option can be placed in any position in command line and takes effect for verification of all successive files in command line. Option always overrides previous occurrence of the same option. Some options specify parameters of global analysis, which is performed after loading of all files, so only the last occurrence of such options takes effect.

Options are always compared ignoring letters case and '_' symbols. So the following two strings specify the same option: *-ShadowLocal* and *-shadow_local*.

All Jlint options are prefixed by '-' or '+'. For options, which can be enabled or disabled, '+' means that option is enabled and '-' means that option is disabled. For options like *source* or *help* there is no difference between '-' and '+'.

'`-source` **path**'
: Specifies path to source files. It is necessary to specify this option when sources and class files are located in different directories. For example: *jlint -source /usr/local/jdk1.1.1/src /usr/local/jdk1.1.1/lib/classes.zip*.

'`-history` **file**'
: Specifies history file. Jlint will not repeatedly report messages, which are present in history file. History file should be available for reading/writing and is appended by new messages after each Jlint execution. This messages will not be more reported in successive executions of Jlint (certainly if *-history* options is present and specifies the same history file).

'`-max_shown_paths` **number**'
: Specifies number of different paths between two vertexes in class graph used for detecting possible deadlocks (See Section 3.1 [Synchronization], page 8). Default value of this parameter is 4. Increasing of this value can increase time of verification for complex programs.

'`-help`'
: Output list of all options, including message categories. If option *+verbose* was previously specified, then list of all messages is also printed.

'`(+-)verbose`'
: Switch on/off verbose mode. In verbose mode Jlint outputs more information about process of verification: names of verified files, warnings about absence of debugging information...

‘`(+-)`**message_category**’
: Enable or disable reporting of messages of specified category. It is possible to disable top level category and then enable some sub-categories within this category. And visa-versa it is possible to disable some specific categories within top-level category. It is also possible to disable concrete message codes within category. Table below describes full hierarchy of messages. By default all categories are enabled.

‘`(+-)all`’
: Enable/disable reporting of all messages. If *-all* is specified, it is possible to enable reporting of some specific categories of messages. For example to output only synchronization messages it is enough to specify "*-all +synchronization*".

‘`(+-)`**message_code**’
: Enable or disable reporting of concrete message. Message will be reported if its category is enabled and message code is enabled. If there is only one message code in the category, then names of the category and message code are the same. By default all messages are enabled.

4.3 Jlint messages hierarchy

5 How to build and use Jlint and AntiC

Jlint is written on C++, using almost no operation system dependent code, so I hope it will not a problem to compile it on any system with C++ compiler. Current release contains makefile for Unix with gcc and for Windows with Microsoft Visual C++. In both cases it is enough to execute "make" to build "antic" and "jlint" programs. Distributive for Windows already includes executable files.

To use Jlint you need to compile first you Java sources to byte code. As far as format of Java class is standard, you can use any available Java compiler. It is preferable to make compiler to include debug information in compiled classes (line table and local variables mapping). In this case Jlint messages will be more detailed. If your are using Sun **javac** compiler, required option is *-g*. Most of compilers by default includes line table, but do not generate local variable table. For example free Java compiler **guavac** can't generate it at all. Some compilers (like Sun's **javac**) can't generate line table if optimization is switch on. If you specify *-verbose* option to Jlint, it will report when it can't find line or local variable table in the class file.

Now Jlint and AntiC produce message in Emacs format: "*file:line: message text*". So it is possible to walk through these messages in Emacs if you start Jlint or AntiC as compiler. You can change prefix *MSG_LOCATION_PREFIX* (defined in '`types.hh`') from "*%0s:%1d:* " to one recognized by your favourite editor or IDE. All Jlint messages are gathered in file '`jlint.msg`', so you can easily change them (but recompilation is needed).

AntiC also includes in the message position in the line. All AntiC messages are produced by function `message_at(int line, int coln, char* msg)`, defined in file '`antic.c`'. You can change format of reported messages by modifying this function.

6 Release notes