A Comparison of Bug Finding Tools for Java[*]

Nick Rutar Christian B. Almazan Jeffrey S. Foster
University of Maryland, College Park
{rutar, almazan, jfoster}@cs.umd.edu

Abstract

Bugs in software are costly and difficult to find and fix. In recent years, many tools and techniques have been developed for automatically finding bugs by analyzing source code or intermediate code statically (at compile time). Different tools and techniques have different tradeoffs, but the practical impact of these tradeoffs is not well understood. In this paper, we apply five bug finding tools, specifically Bandera, ESC/Java 2, FindBugs, JLint, and PMD, to a variety of Java programs. By using a variety of tools, we are able to cross-check their bug reports and warnings. Our experimental results show that none of the tools strictly subsumes another, and indeed the tools often find non-overlapping bugs. We discuss the techniques each of the tools is based on, and we suggest how particular techniques affect the output of the tools. Finally, we propose a meta-tool that combines the output of the tools together, looking for particular lines of code, methods, and classes that many tools warn about.

1 Introduction

In recent years, many tools have been developed for automatically finding bugs in program source code, using techniques such as syntactic pattern matching, data flow analysis, type systems, model checking, and theorem proving. Many of these tools check for the same kinds of programming mistakes, yet to date there has been little direct comparison between them. In this paper, we perform one of the first broad comparisons of several Java bug-finding tools over a wide variety of tasks.

In the course of our experiments, we discovered, somewhat surprisingly, that there is clearly no single "best" bug-finding tool. Indeed, we found a wide range in the kinds of bugs found by different tools (Section 2). Even in the cases when different tools purport to find the same kind of bug, we found that in fact they often report different instances of the bug in different places (Section 4.1). We also found that many tools produce a large volume of warnings, which makes it hard to know which to look at first.

Even though the tools do not show much overlap in particular warnings, we initially thought that they might be correlated overall. For example, if one tool issues many warnings for a class, then it might be likely that another tool does as well. However, our results show that this is not true in general. There is no correlation of warning counts between pairs of tools. Additionally, and perhaps surprisingly, warning counts are not strongly correlated with lines of code.

Given these results, we believe there will always be a need for many different bug finding tools, and we propose creating a *bug finding meta-tool* for automatically combining and correlating their output (Section 3). Using this tool, developers can look for code that yields an unusual number of warnings from many different tools. We explored two different metrics for using warning counts to rank code as suspicious, and we discovered that both are correlated for the highest-ranked code (Section 4.2).

For our study, we selected five well-known, publicly available bug-finding tools (Section 2.2). Our study focuses on PMD [18], FindBugs [13], and JLint [16], which use syntactic bug pattern detection. JLint and FindBugs also include a dataflow component. Our study also includes ESC/Java [10], which uses theorem proving, and Bandera [6], which uses model checking.

We ran the tools on a small suite of variously-sized Java programs from various domains. It is a basic undecidability result that no bug finding tool can always report correct results. Thus all of the tools must balance finding true bugs with generating *false positives* (warnings about correct code) and *false negatives* (failing to warn about incorrect code). All of the tools make different tradeoffs, and these choices are what cause the tools to produce the wide range of results we observed for our benchmark suite.

The main contributions of this paper are as follows:

- We present what we believe is the first detailed comparison of several different bug finding tools for Java over a variety of checking tasks.

[*] This research was supported in part by NSF CCF-0346982.

- We show that, even for the same checking task, there is little overlap in the warnings generated by the tools. We believe this occurs because all of the tools choose different tradeoffs between generating false positives and false negatives.
- We also show that the warning counts from different tools are not generally correlated. Given this result, we believe that there will always be a need for multiple separate tools, and we propose a bug finding meta-tool for combining the results of different tools together and cross-referencing their output to prioritize warnings. We show that two different metrics tend to rank code similarly.

1.1 Threats to Validity

There are a number of potential threats to the validity of this study. Foremost is simply the limited scope of the study, both in terms of the test suite size and in terms of the selection of tools. We believe, however, that we have chosen a representative set of Java benchmarks and Java bug finding tools. Additionally, there may be other considerations for tools for languages such as C and C++, which we have not studied. However, since many tools for those languages use the same basic techniques as the tools we studied, we think that the lessons we learned will be applicable to tools for those languages as well.

Another potential threat to validity is that we did not exactly categorize every false positive and false negative from the tools. Doing so would be extremely difficult, given the large number of warnings from the tools and the fact that we ourselves did not write the benchmark programs in our study. Instead, in Section 4.1, we cross-check the results of the tools with each other in order to get a general sense of how accurate the warnings are, and in order to understand how the implementation techniques affect the generated warnings. We leave it as interesting future work to check for false negatives elsewhere, e.g., in CVS revision histories or change logs.

A final threat to validity is that what we make no distinction between the severity of one bug versus another. Quantifying the severity of bugs is a difficult problem, and it is not the focus of this paper. For example, consider the following piece of code:

```
int x = 2, y = 3;
if (x == y)
   if (y == 3)
       x = 3;
else
   x = 4;
```

In this example, indentation would suggest that the `else` corresponds to the first `if`, but the language grammar says otherwise. The result is most likely a logical error, since a programmer might believe this code will result in `x=4` when it really results in `x=2`. Depending on later uses of `x`, this could be a major error. Used with the right rulesets for ensuring that all `if` statements use braces around the body, PMD will flag this program as suspicious.

The following more blatant error is detected by JLint, FindBugs, and ESC/Java:

```
String s = new String("I'm not null...yet");
s = null;
System.out.println(s.length());
```

This segment of code will obviously cause an exception at runtime, which is not desirable, but will have the effect of halting the program as soon as the error occurs (assuming the exception is not caught). Moreover, if it is on a common program path, this error will most likely be discovered when the program is run, and the exception will pinpoint the exact location of the error.

When asked which is the more severe bug, many programmers might say that a null dereference is worse than not using braces in an `if` statement (which is often not an error at all). And yet the logical error caused by the lack of braces might perhaps be much more severe, and harder to track down, than the null dereference.

These small examples illustrate that for any particular program bug, the severity of the error cannot be separated from the context in which the program is used. With this in mind, in Section 6 we mention a few ways in which user-specified information about severity might be taken into account.

2 Background

2.1 A Small Example

The code sample in Figure 1 illustrates the variety and typical overlap of bugs found by the tools. It also illustrates the problems associated with false positives and false negatives. The code in Figure 1 compiles with no errors and no warnings, and though it won't win any awards for functionality, it could easily be passed off as fine. However, four of the five tools were each able to find at least one bug in this program. (Bandera wasn't tested against the code for reasons explained later.)

PMD discovers that the variable `y` on line 8 is never used and generates an "Avoid unused local variables" warning. FindBugs displays a "Method ignores results of InputStream.read()" warning for line 12; this is an error because the result of `InputStream.read()` is the number of bytes read, and this may be fewer bytes than the programmer is expecting. FindBugs also displays a "Method

```
1   import java.io.*;
2   public class Foo{
3     private byte[] b;
4     private int length;
5     Foo(){ length = 40;
6       b = new byte[length]; }
7     public void bar(){
8       int y;
9       try {
10        FileInputStream x =
11           new FileInputStream("z");
12        x.read(b,0,length);
13        x.close();}
14      catch(Exception e){
15        System.out.println("Oopsie");}
16      for(int i = 1; i <= length; i++){
17        if (Integer.toString(50) ==
18                 Byte.toString(b[i]))
19          System.out.print(b[i] + " ");
20      }
21    }
22  }
```

Figure 1. A Sample Java Class

may fail to close stream on exception” warning and a warning on lines 17-18 for using “==” to compare String objects (which is incorrect). ESC/Java displays “Warning: Array index possibly too large” because the comparison on lines 17-18 may access an element outside the bounds of the array due to an error in the loop guard on line 16. This, as we can see, is a valid error. However, ESC/Java also displays a warning for “Possible null dereference” on line 18, which is a false positive since `b` is initialized in the constructor. Finally, JLint displays a “Compare strings as object references” warning for the string comparison on lines 17-18. This is the same error that FindBugs detected, which illustrates that there is some overlap between the tools.

This is the sum total of all the errors reported by the tools. The results for this small example also illustrate several cases of false negatives. FindBugs, for instance, also looks for unused variables, but does not discover that the variable `y` on line 8 was never used. As another example, JLint sometimes warns about indexing out of bounds, but JLint does not recognize the particular case on line 16. Further examples of overlapping warnings between programs, false positives, and false negatives are all discussed later on in the paper.

2.2 Java Bug Finding Tools

Figure 2 contains a brief summary of the five tools we study in this paper, and below we discuss each of them in more detail.

Name	Version	Input	Interfaces	Technology
Bandera	0.3b2 (2003)	Source	CL, GUI	Model checking
ESC/Java	2.0a7 (2004)	Source[1]	CL, GUI	Theorem proving
FindBugs	0.8.2 (2004)	Bytecode	CL, GUI, IDE, Ant	Syntax, dataflow
JLint	3.0 (2004)	Bytecode	CL	Syntax, dataflow
PMD	1.9 (2004)	Source	CL, GUI, Ant, IDE	Syntax

CL - Command Line
[1] ESC/Java works primarily with source but may require bytecode or specification files for supporting types.

Figure 2. Bug Finding Tools and Their Basic Properties

FindBugs [13] is a bug pattern detector for Java. FindBugs uses a series of ad-hoc techniques designed to balance precision, efficiency, and usability. One of the main techniques FindBugs uses is to syntactically match source code to known suspicious programming practice, in a manner similar to ASTLog [7]. For example, FindBugs checks that calls to `wait()`, used in multi-threaded Java programs, are always within a loop—which is the correct usage in most cases. In some cases, FindBugs also uses dataflow analysis to check for bugs. For example, FindBugs uses a simple, intraprocedural (within one method) dataflow analysis to check for null pointer dereferences.

FindBugs can be expanded by writing custom bug detectors in Java. We set FindBugs to report “medium” priority warnings, which is the recommended setting.

JLint [1, 16], like FindBugs, analyzes Java bytecode, performing syntactic checks and dataflow analysis. JLint also includes an interprocedural, inter-file component to find deadlocks by building a lock graph and ensuring that there are never any cycles in the graph. JLint 3.0, the version we used, includes the multi-threaded program checking extensions described by Artho [1]. JLint is not easily expandable.

PMD [18], like FindBugs and JLint, performs syntactic checks on program source code, but it does not have a dataflow component. In addition to some detection of clearly erroneous code, many of the “bugs” PMD looks for are stylistic conventions whose violation might be suspicious under some circumstances. For example, having a `try` statement with an empty `catch` block might indicate that the caught error is incorrectly discarded. Because PMD includes many detectors for bugs that depend on programming style, PMD includes support for selecting which detectors or groups of detectors should be run. In our experiments, we run PMD with the rulesets recommended by the documentation: unusedcode.xml, basic.xml, import.xml, and favorites.xml. The number of warnings can increase or decrease depending on which rulesets are used. PMD is easily extensible by programmers, who can write new bug pattern detectors using either Java or XPath.

Bandera [6] is a verification tool based on model checking and abstraction. To use Bandera, the programmer annotates their source code with specifications describing what should be checked, or no specifications if the programmer only wants to verify some standard synchronization properties. In particular, with no annotations Bandera verifies the absence of deadlocks. Bandera includes optional slicing and abstraction phases, followed by model checking. Bandera can use a variety of model checkers, including SPIN [12] and the Java PathFinder [11].

We included Bandera in our study because it uses a completely different technique than the other tools we looked at. Unfortunately, Bandera version 0.3b2 does not run on any realistic Java programs, including our benchmark suite. The developers of Bandera acknowledge on their web page that it cannot analyze Java (standard) library calls, and unfortunately the Java library is used extensively by all of our benchmarks. This greatly limits the usability and applicability of Bandera (future successors will address this problem). We were able to successfully run Bandera and the other tools on the small example programs supplied with Bandera. Section 5 discusses the results.

ESC/Java [10], the Extended Static Checking system for Java, based on theorem proving, performs formal verification of properties of Java source code. To use ESC/Java, the programmer adds preconditions, postconditions, and loop invariants to source code in the form of special comments. ESC/Java uses a theorem prover to verify that the program matches the specifications.

ESC/Java is designed so that it can produce some useful output even without any specifications, and this is the way we used it in our study. In this case, ESC/Java looks for errors such as null pointer dereferences, array out-of-bounds errors, and so on; annotations can be used to remove false positives or to add additional specifications to be checked.

For our study, we used ESC/Java 2 [5], a successor to the original ESC/Java project. ESC/Java 2 includes support for Java 1.4, which is critical to analyzing current applications. ESC/Java 2 is being actively developed, and all references to ESC/Java will refer to the ESC/Java 2, rather than the original ESC/Java.

We included ESC/Java in our set of tools because its approach to finding bugs is notably different from the other tools. However, as we will discuss in Section 3, without annotations ESC/Java produces a multitude of warnings. Houdini [9] can automatically add ESC/Java annotations to programs, but it does not work with ESC/Java 2 [4]. Daikon [8] can also be used as an annotation assistant to ESC/Java, but doing so would require selecting representative dynamic program executions that sufficiently cover the program paths, which we did not attempt. Since ESC/Java really works best with annotations, in this paper we will mostly use it as a point of comparison and do not include it in the meta-tool metrics in Section 4.2.

2.3 Taxonomy of Bugs

We classified all of the bugs the tools find into the groups listed in Figure 3. The first column lists a general class of bugs, and the second column gives one common example from that class. The last columns indicate whether each tool finds bugs in that category, and whether the tools find the specific example we list. We did not put Bandera in this table, since without annotations its checks are limited to synchronization properties.

These classifications are our own, not the ones used in the literature for any of these tools. With this in mind, notice that the largest overlap is between FindBugs and PMD, which share 6 categories in common. The “General” category is a catch-all for checks that do not fit in the other categories, so all tools find something in that category. All of the tools also look for concurrency errors. Overall, there are many common categories among the tools and many categories on which the tools differ.

Other fault classifications that have been developed are not appropriate for our discussion. Two such classifications, the Orthogonal Defect Classification [3] and the IEEE Standard Classification for Software Anomalies [14], focus on the overall software life cycle phases. Both treat faults at a much higher-level than we do in this paper. For example, they have a facility for specifying that a fault is a logic problem, but do not provide specifications for what the logic problem leads to or was caused by, such as incorrect synchronization.

3 Experiments

To generate the results in this paper, we wrote a series of scripts that combine and coordinate the output from the various tools. Together, these scripts form a preliminary version of the bug finding *meta-tool* that we mentioned in the introduction. This meta-tool allows a developer to examine the output from all the tools in a common format and find what classes, methods, and lines generate warnings.

As discussed in the introduction, we believe that such a meta-tool can provide much better bug finding ability than

Bug Category	**Example**	ESC/Java	FindBugs	JLint	PMD
General	Null dereference	√*	√*	√*	√
Concurrency	Possible deadlock	√*	√	√*	√
Exceptions	Possible unexpected exception	√*			
Array	Length may be less than zero	√		√*	
Mathematics	Division by zero	√*		√	
Conditional, loop	Unreachable code due to constant guard		√		√*
String	Checking equality using == or !=		√	√*	√
Object overriding	Equal objects must have equal hashcodes		√*	√*	√*
I/O stream	Stream not closed on all paths		√*		
Unused or duplicate statement	Unused local variable		√		√*
Design	Should be a static inner class		√*		
Unnecessary statement	Unnecessary return statement				√*

√ - tool checks for bugs in this category   * - tool checks for this specific example

Figure 3. The Types of Bugs Each Tool Finds

the tools in isolation. As Figure 3 shows, there is a lot of variation even in the kinds of bugs found by the tools. Moreover, as we will discuss in Section 4.1, there are not many cases where multiple tools warn about the same potential problem. Having a meta-tool means that a developer need not rely on the output of a single tool. In particular, the meta-tool can rank classes, methods, and lines by the number of warnings generated by the various tools. In Section 4.2, we will discuss simple metrics for doing so and examine the results.

Of course, rather than having a meta-tool, perhaps the ideal situation would be a single tool with many different analyses built-in, and the different analyses could be combined and correlated in the appropriate fashion. However, as a practical matter, the tools tend to be written by a wide variety of developers, and so at least for now having a separate tool to combine their results seems necessary.

The preliminary meta-tool we built for this paper is fairly simple. Its main tasks are to parse the different textual output of the various tools (ranging from delimited text to XML) and map the warnings, which are typically given by file and line, back to classes and methods. We computed the rankings in a separate pass. Section 6 discusses some possible enhancements to our tool.

We selected as a testbed five mid-sized programs compiled with Java 1.4. The programs represent a range of applications, with varying functionality, program size, and program maturity. The five programs are

Apache Tomcat 5.019 Java Servlet and JavaServer Pages implementation, specifically catalina.jar[1]

JBoss 3.2.3 J2EE application server[2]

Art of Illusion 1.7 3D modeling and rendering studio[3]

Azureus 2.0.7 Java Bit Torrent client[4]

Megamek 0.29 Online version of BattleTech game[5]

Figure 4 lists the size of each benchmark in terms of both Non Commented Source Statements (NCSS), roughly the number of ';' and '{' characters in the program, and the number of class files. The remaining columns of Figure 4 list the running times and total number of warnings generated by each tool. Section 4 discusses the results in-depth; here we give some high-level comments. Bandera is not included in this table, since it does not run on any of these examples. See Section 5.

To compute the running times, we ran all of the programs from the command line, as the optional GUIs can potentially reduce performance. Execution times were computed with one run, as performance is not the emphasis of this study. The tests were performed on a Mac OS X v10.3.3 system with a 1.25 GHz PowerPC G4 processor and 512 MB RAM. Because PMD accepts only one source file at a time, we used a script to invoke it on every file in each benchmark. Unfortunately, since PMD is written in Java, each invocation launches the Java virtual machine separately, which significantly reduces PMD's performance. We expect that without this overhead, PMD would be approximately 20% faster. Recall that we used ESC/Java without annotations; we do not know if adding annotations would affect ESC/Java's running time, but we suspect it will still run significantly slower than the other tools. Speaking in general terms, ESC/Java takes a few hours to run, FindBugs and PMD take a few minutes, and JLint takes a few seconds.

For each tool, we report the absolute number of warnings generated, with no normalization or attempt to discount repeated warnings about the same error. Thus we are measuring the total volume of information presented to a de-

[1] `http://jakarta.apache.org/tomcat`

[2] `http://www.jboss.org`

[3] `http://www.artofillusion.org`

[4] `http://azureus.sourceforge.net`

[5] `http://megamek.sourceforge.net`

Name	NCSS	Class	Time (min:sec.csec)				Warning Count			
	(Lines)	Files	ESC/Java	FindBugs	JLint	PMD	ESC/Java	FindBugs	JLint	PMD
Azureus 2.0.7	35,549	1053	211:09.00	01:26.14	00:06.87	19:39.00	5474	360	1584	1371
Art of Illusion 1.7	55,249	676	361:56.00	02:55.02	00:06.32	20:03.00	12813	481	1637	1992
Tomcat 5.019	34,425	290	90:25.00	01:03.62	00:08.71	14:28.00	1241	245	3247	1236
JBoss 3.2.3	8,354	274	84:01.00	00:17.56	00:03.12	09:11.00	1539	79	317	153
Megamek 0.29	37,255	270	23:39.00	02:27.21	00:06.25	11:12.00	6402	223	4353	536

Figure 4. Running Time and Warnings Generated by Each Tool

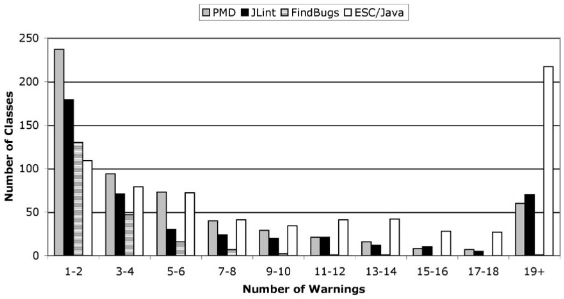

Figure 5. Histogram for number of warnings found per class

	ESC/ Java	Find Bugs	JLint	PMD
Concurrency Warnings	126	122	8883	0
Null Dereferencing	9120	18	449	0
Null Assignment	0	0	0	594
Index out of Bounds	1810	0	264	0
Prefer Zero Length Array	0	36	0	0

Figure 6. Warning Counts for the Categories Discussed in Section 4.1

veloper from each tool. For ESC/Java, the number of generated warnings is sometimes extremely high. Among the other tools, JLint tends to report the largest number of warnings, followed by PMD (though for Art of Illusion, PMD reported more warnings than JLint). FindBugs generally reports fewer warnings than the other tools. In general, we found this makes FindBugs easier to use, because there are fewer results to examine.

Figure 5 shows a histogram of the warning counts per class. (We do not include classes with no warnings.) Clearly, in most cases, when the tools find potential bugs, they only find a few, and the number of classes with multiple warnings drops off rapidly. For PMD and JLint, there are quite a few classes that have 19 or more warnings, while these are rare for FindBugs. For ESC/Java, many classes have 19 or more warnings.

4 Analysis

4.1 Overlapping Bug Categories

Clearly the tools generate far too many warnings to review all of them manually. In this section, we examine the effectiveness of the tools on three checking tasks that several of the tools share in common: concurrency, null dereference, and array bounds errors. Even for the same task we found a wide variation in the warnings reported by different tools. Figure 6 contains a breakdown of the warning counts. Even after restricting ourselves to these three categories, there is still a large number of warnings, and so our manual examination is limited to several dozen warnings.

Concurrency Errors All of the tools check for at least one kind of concurrency error. ESC/Java includes support for automatically checking for race conditions and potential deadlocks. ESC/Java finds no race conditions, but it issues 126 deadlock warnings for our benchmark suite. After investigating a handful of these warnings, we found that some of them appear to be false positives. Further investigation is difficult, because ESC/Java reports `synchronized` blocks that are involved in potential deadlocks but not the sets of locks in each particular deadlock.

PMD includes checks for some common bug patterns, such as the well-known double-checked locking bug in Java [2]. However, PMD does not issue any such warnings for our benchmarks. In contrast, both FindBugs and JLint do report warnings. Like PMD, FindBugs also checks for uses of double-checked locking. Interestingly, despite PMD reporting no such cases, FindBugs finds a total of three uses of double-checked locking in the benchmark programs. Manual examination of the code shows that, indeed, those three uses are erroneous. PMD does not report this error because its checker is fooled by some other code mixed in with the bug pattern (such as try/catch blocks).

FindBugs also warns about the presence of other concurrency bug patterns, such as not putting a monitor `wait()` call in a while loop. Examining the results in detail, we discovered that the warnings FindBugs reports usually correctly indicate the presence of the bug pattern in the code.

What is less clear is how many of the patterns detected correspond to actual errors. For example, since FindBugs does not perform interprocedural analysis (it analyzes a single method at a time), if a method with a `wait()` is itself called in a loop, FindBugs will still report a warning (though this did not happen in our benchmarks). And, of course, not all uses of `wait()` outside of a loop are incorrect.

On our test suite, JLint generates many warnings about potential deadlocks. In some cases, JLint produces many warnings for the same underlying bug. For instance, JLint checks for deadlock by producing a lock graph and looking for cycles. In several cases in our experiments, JLint iterates over the lock graph repeatedly, reporting the same cycle many times. In some cases, the same cycle generated several hundred warnings. These duplicates, which make it difficult to use the output of JLint, could be eliminated by reporting a cycle in the lock graph just once. The sheer quantity of output from JLint makes it difficult to judge the rate of false positives for our benchmark suite. In Section 5 we compare finding deadlocks using JLint and Bandera on smaller programs.

Null Dereferences Among the four tools, ESC/Java, FindBugs, and JLint check for null dereferences. Surprisingly, there is not a lot of overlap between the warnings reported by the various tools.

JLint finds many potential null dereferences. In order to reduce the number of warnings, JLint tries to only identify inconsistent assumptions about null. For example, JLint warns if an object is sometimes compared against null before it is dereferenced and sometimes not. However, we have found that in a fair number of cases, JLint's null dereference warnings are false positives. A common example is when conditional tests imply that an object cannot be null (e.g., because it was not null previously when the condition held). In this case, JLint often does not track enough information about conditionals to suppress the warning. Finally, in some cases there are warnings about null pointer dereferences that cannot happen because of deeper program logic; not many static analyses could handle these cases. Currently, there is no way to stop these warnings from being reported (sometimes multiple times).

ESC/Java reports the most null pointer dereferences because it often assumes objects might be null, since we did not add any annotations to the contrary. (Interestingly, ESC/Java does not always report null dereference warnings in the same places as JLint). The net result is that, while potentially those places may be null pointer errors, there are too many warnings to be easily useful by themselves. Instead, to make the most effective use of these checks, it seems the programmer should provide annotations. For example, in method declarations parameters that are never null can be marked as such to avoid spurious warnings.

Interestingly, FindBugs discovers a very small set of potential null dereferences compared to both ESC/Java and JLint. This is because FindBugs uses several heuristics to avoid reporting null-pointer dereference warnings in certain cases when its dataflow analysis loses precision.

PMD does not check for null pointer dereferences, but it does warn about setting certain objects to null. We suspect this check is not useful for many common coding styles. ESC/Java also checks for some other uses of null that violate implicit specifications, e.g., assigning null to a field assumed not to be null. In a few cases, we found that PMD and ESC/Java null warnings coincide with each other. For example, in several cases PMD reported an object being set to null, and just a few lines later ESC/Java issued a warning about assigning null to another object.

Array Bounds Errors In Java, indexing outside the bounds of an array results is a run-time exception. While a bounds error in Java may not be the catastrophic error that it can be for C and C++ (where bounds errors overwrite unexpected parts of memory), they still indicate a bug in the program. Two of the tools we examined, JLint and ESC/Java, include checks for array bounds errors—either creating an array with a negative size, or accessing an array with an index that is negative or greater than the size of the array.

Like null dereference warnings, JLint and ESC/Java do not always report the same warnings in the same places. ESC/Java mainly reports warnings because parameters that are later used in array accesses may not be within range (annotations would help with this). JLint has several false positives and some false negatives in this category, apparently because it does not track certain information interprocedurally in its dataflow analysis. For example, code such as this appeared in our benchmarks:

```
public class Foo {
  static Integer[] ary = new Integer[2];

  public static void assign() {
    Object o0 = ary[ary.length];
    Object o1 = ary[ary.length-1];
  }
}
```

In this case, JLint signals a warning that the array index might be out of bounds for the access to `o1` (because it thinks the length of the array might be 0), but clearly that is not possible here. On the other hand, there are no warnings for the access to `o0`, even though it will always be out of bounds no matter what size the array is.

FindBugs and PMD do not check for array bounds errors, though FindBugs does warn about returning null from

Tools	Correlation coefficient
JLint vs PMD	0.15
JLint vs FindBugs	0.33
FindBugs vs PMD	0.31

Figure 7. Correlation among Warnings from Pairs of Tools

a method that returns an array (it may be better to use a 0-length array).

4.2 Cross-Tool Buggy Code Correlations

When initially hypothesizing about the relationship among the tools, we conjectured that warnings among the different tools were correlated, and that the meta-tool would show that more warnings from one tool would correspond to more warnings from other tools. However, we found that this is not necessarily the case. Figure 7 gives the correlation coefficients for the number of warnings found by pairs of tools per class. As these results indicate, the large number of warnings reported by some tools are sometimes simply anomalous, and there does not seem to be any general correlation between the total number of warnings one tools generates and the total number of warnings another tool generates for any given class.

We also wanted to check whether the number of warnings reported is simply a function of the number of lines of code. Figure 8 gives correlation coefficients and scatter plots showing, for each Java source file (which may include several inner classes), the NCSS count versus the number of warnings. For JLint, we have removed from the chart five source files that had over 500 warnings each, since adding these makes it hard to see the other data points. As these plots show, there does not seem to be any general correlation between lines of code and number of warnings produced by any of the tools. JLint has the strongest correlation of the three, but it is still weak.

4.2.1 Two Simple Metrics for Isolating Buggy Code

Given that the tools' warnings are not generally correlated, we hypothesize that combining the results of multiple tools together can identify potentially troublesome areas in the code that might be missed when using the tools in isolation. Since we do not have exhaustive information about the severity of faults identified by the warnings and rates of false positives and false negatives, we cannot form any strong conclusions about the benefit of our metrics. Thus in this section we perform only a preliminary investigation.

We studied two metrics for ranking code. As mentioned in Section 2.2, we do not include ESC/Java in this discussion.

For the first metric, we started with the number of warnings per class file from each tool. (The same metric can also be used per method, per lexical scope, or per line.) For a particular benchmark and a particular tool, we linearly scaled the per-class warning counts to range between 0 and 1, with 1 being the maximum number of per-class warning counts reported by the tool over all our benchmarks.

Formally, let n be the total number of classes, and let X_i be the number of warnings reported by tool X for class number i, where $i \in 1..n$. Then we computed a normalized warning count

$$X'_i = X_i / \max_{i=1}^{n} X_i$$

Then for class number i, we summed the normalized warning counts from each tool to compute our first metric, the *normalized warning total*:

$$Total_i = FindBugs_i + JLint_i + PMD_i$$

In order to avoid affecting the scaling for JLint, we reduced its warning count for the class with the highest number of errors from 1979 to 200, and for the next four highest classes to 199 through 196, respectively (to maintain their ranking)

With this first metric, the warning counts could be biased by repeated warnings about the same underlying bug. In order to compensate for this possibility, we developed a second metric, the *unique warning total*, that counts only the first instance of each type of warning message generated by a tool. For example, no matter how many null pointer dereferences FindBugs reports in a class, we only count this as 0 (if none were found) or 1 (if one or more were found). In this metric, we sum the number of unique warnings from all the tools.

4.2.2 Results

We applied these metrics to our benchmark suite, ranking the classes according to their normalized and unique warning totals. As it turns out, these two metrics are fairly well correlated, especially for the classes that are ranked highest by both metrics. Figure 9 shows the relationship between the normalized warning count and the number of unique warnings per class. The correlation coefficient for this relationship is 0.758. Of course, it is not surprising that these metrics are correlated, because they are clearly not independent (in particular, if one is non-zero then the other must be as well). However, a simple examination of certain classes shows that the high correlation coefficient between the two is not obvious. For instance, the class catalina.context has a warning count of 0 for FindBugs and JLint, but PMD generates 132 warnings. (As it turns out, PMD's warnings are

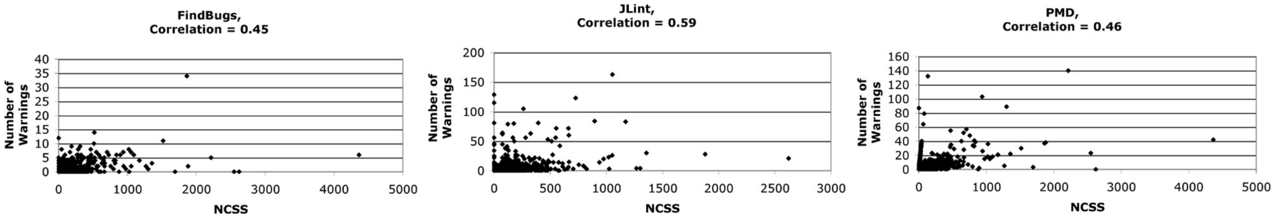

Figure 8. Comparison of Number of Warnings versus NCSS

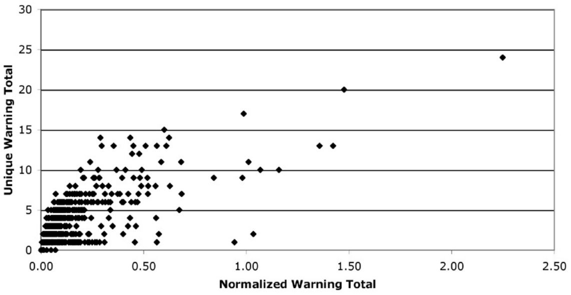

Figure 9. Normalized Warnings versus the Unique Warnings per Class

uninteresting). This class ranks 11th in normalized warning total, but 587th in unique warning total (all 132 warnings are the same kind). Thus just because a class generates a large number of warnings does not necessarily mean that it generates a large breath of warnings.

We manually examined the warnings for the top five classes for both metrics, listed in Figure 10. For these classes, Figure 10 shows the size of the class, in terms of NCSS and number of methods, the normalized warning total and rank, the total number of warnings found by each of the tools, and the number of unique warnings and rank. In this table, T-n denotes a class ranked n, which is tied with at least one other class in the ranking.

Recall that the goal of our metrics is to identify code that might be missed when using the tools in isolation. In this table, the top two classes in both metrics are the same, catalina.core.StandardContext and megamek.server.Server, and both also have the most warnings of any class from, respectively, FindBugs and JLint. Thus these classes, as well as artofillusion.object.TriangleMesh (with the most warnings from PMD), can be identified as highly suspicious by a single tool.

On the other hand, azureus2.ui.swt.MainWindow could be overlooked when considering only one tool at a time. It is ranked in the top 10 for both of our metrics, but it is 4th for FindBugs in isolation, 13th for JLint, and 30th for PMD. As another example, catalina.core.StandardWrapper (4th for the unique warning metric), is ranked 45th for FindBugs, 11th for JLint, and 349th for PMD—thus if we were only using a single tool, we would be unlikely to examine the warnings for it immediately.

In general, the normalized warning total measures the number of tools that find an unusually high number of warnings. The metric is still susceptible, however, to cases where a single tool produces a multitude of spurious warnings. For example, megamek.server.Server has hundreds of null dereference warnings from JLint, many likely false positives, which is why it is ranked second in this metric. In the case of artofillusion.object.TriangleMesh, 102 out of 140 of the warnings from PMD are for not using brackets in a `for` statement—which it probably not a mistake at all.

On the other hand, the unique warning total measures the breadth of warnings found by the tools. This metric compensates for cascading warnings of the same kind, but it can be fooled by redundancy among the different tools. For example, if by luck a null deference error is caught by two separate tools, then the warning for that error will be counted twice. This has a large affect on the unique warning counts, because they are in general small. An improved metric could solve this problem by counting uniqueness of errors across all tools (which requires identifying duplicate messages across tools, a non-obvious task for some warnings that are close but not identical).

We think that both metrics provide a useful gauge that allows programmers to go beyond finding individual bugs with individual tools. Instead, these metrics can be used to find code with an unusually high number and breadth of warnings from many tools—and our results show that both seem to be correlated for the highest-ranked classes.

5 Bandera

Bandera cannot analyze any of our benchmarks from Section 3, because it cannot analyze the Java library. In order to compare Bandera to the other tools, we used the small examples supplied with Bandera as a test suite, since we knew that Bandera could analyze them.

This test suite from Bandera includes 16 programs ranging from 100-300 lines, 8 of which contain a real deadlock. None of the programs include specifications—without specifications, Bandera will automatically check for deadlock.

			Total Warnings			Normalized		Unique Warnings				
Name	NCSS	Mthds	FB	JL	PMD	$Total_i$	Rank	FB	JL	PMD	Total	Rank
catalina.core.StandardContext	1863	255	*34	791	37	2.25	1	9	10	5	24	1
megamek.server.Server	4363	198	6	*1979	42	1.48	2	6	10	4	20	2
azureus2.ui.swt.MainWindow	1517	87	11	90	30	0.99	9	5	8	4	17	3
catalina.core.StandardWrapper	513	75	10	50	8	0.60	19	6	6	3	15	4
catalina.core.StandardHost	279	55	4	97	3	0.62	17	10	3	1	14	5
catalina.core.ContainerBase	518	70	14	849	3	1.42	3	3	7	3	13	T-8
artofillusion.object.TriangleMesh	2213	59	5	42	*140	1.36	4	3	7	3	13	T-8
megamek.common.Compute	2250	109	0	1076	23	1.16	5	0	7	3	10	T-22

* - Class with highest number of warnings from this tool

Figure 10. Classes Ranked Highly by Metrics

For this test suite, Bandera finds all 8 deadlocks and produces no messages concerning the other 8 programs.

In comparison, FindBugs and PMD do not issue any warnings that would indicate a deadlock. PMD reports 19 warnings, but only about null assignments and missing braces around loop bodies, which in this case has no effect on synchronization. FindBugs issues 5 warnings, 4 of which were about package protections and the other of which warned about using `notify()` instead of `notifyAll()` (the use of `notify()` is correct).

On the other hand, ESC/Java reports 79 warnings, 30 of which are for potential deadlocks in 9 of the programs. One of the 9 programs did not have deadlock. JLint finds potential synchronization bugs in 5 of the 8 programs Bandera verified to have a deadlock error. JLint issues three different kinds of concurrency warnings for these programs: a warning for changing a lock variable that has been used in synchronization, a warning for requesting locks that would lead to a lock cycle, and a warning for improper use of monitor objects. In all, JLint reported 34 potential concurrency bugs over 5 programs.

Compared to JLint, Bandera has the advantage that it can produce counterexamples. Because it is based on model checking technology, when Bandera finds a potential deadlock it can produce a full program trace documenting the sequence of operations leading up to the error and a graphical representation of the lock graph with a deadlock. Non-model checking tools such as JLint often are not as well geared to generating counterexample traces.

6 Usability Issues and Improvements to the Meta-Tool

In the course of our experiments, we encountered a number of issues in applying the tools to our benchmark suite. Some of these issues must be dealt with within a tool, and some of the issues can be addressed by improving our proposed meta-tool.

In a number of cases, we had difficulty using certain versions of the tools because they were not compatible with the latest version of Java. As mentioned earlier, when we initially experimented with ESC/Java, we downloaded version 0.7 and discovered that it was not compatible with Java 1.4. Fortunately ESC/Java 2, which has new developers, is compatible, so we were able to use that version for our experiments. But we are still unable to use some important other relations of ESC/Java such as Houdini, which is not compatible with ESC/Java 2. We had similar problems with an older version of JLint, which also did not handle Java 1.4. The lesson for users is probably to rely only on tools under active development, and the lesson for tool builders is to keep up with the latest language features lest a tool become unusable. This may especially be an issue with the upcoming Java 1.5, which includes source-level extensions such as generics.

In our opinion, tools that provide graphical user interfaces (GUIs) or plugins for a variety of integrated development environments have a clear advantage over those tools that provide only textual output. A well-designed GUI can group classes of bugs together and hyperlink warnings to source code. Although we did not use them directly in our study, in our initial phase of learning to use the tools we found GUIs invaluable. Unfortunately, GUIs conflict somewhat with having a meta-tool, since they make it much more difficult for a meta-tool to extract the analysis results. Thus probably the best compromise is to provide both structural text output (for processing by the meta-tool) and a GUI. We leave as future work the development of a generic, easy-to-use GUI for the meta-tool itself.

Also, while developers want to find as many bugs as possible, it is important not to overwhelm the developer with too much output. In particular, one critical ability is to avoid cascading errors. For example, in some cases JLint repeatedly warns about dereferencing a variable that may be null, but it would be sufficient to warn only at the first dereference. These cases may be possible to eliminate with the meta tool. Or, better yet, the tool could be modified so that once a warning about a null pointer is issued, the pointer would subsequently be assumed not to be null (or whatever the most optimistic assumption is) to suppress further warnings. Similarly, sometimes JLint produces a large number of potential deadlock warnings, even reporting the same warning multiple times on the same line. In this case, the meta-cool could easily filter redundant error messages and reduce them to a single warning. In general, the meta-tool could allow the user to select between full output from each of the tools and output limited to unique warnings.

As mentioned throughout this paper, false positives are an issue with all of the tools. ESC/Java is the one tool that supports user-supplied annotations to eliminate spurious warnings. We could incorporate a poor-man's version of this annotation facility into the meta-tool by allowing the user to suppress certain warnings at particular locations in the source code. This would allow the user to prune the output of tools to reduce false positives. In general such a facility must be used extremely carefully, since it is likely that subsequent code modifications might render the suppression of warnings confusing or even incorrect.

A meta-tool could also interpret the output of the tools in complex ways. In particular, it could use a warning from one tool to decide whether another tool's warning has a greater probability of being valid. For example, in one case we encountered, a PMD-generated warning about a null assignment coincided with a JLint warning for the same potential bug. After a manual check of the bug, we found that both tools were correct in their assessment.

Finally, as discussed in Section 1.1, it is impossible to generally classify the severity of a warning without knowing the context in which the application is used. However, it might be possible for developers to classify bug severity for their own programs. Initially, warnings would be weighed evenly, and a developer could change the weights so that different bugs were weighed more or less in the meta-tool's rankings. For example, warnings that in the past have lead to severe errors might be good candidates for increased weight. Weights could also be used to adjust for false positive rates. If a particular bug checker is known to report many false positives for a particular application, those warnings can be assigned a lower weight.

7 Related Work

Artho [1] compares several dynamic and static tools for finding errors in multi-threaded programs. Artho compares the tools on several small core programs extracted from a variety of Java applications. Artho then proposes extensions to JLint, included in the version we tested in this paper, to greatly improve its ability to check for multi-threaded programming bugs, and gives results for running JLint on several large applications. The focus of this paper, in contrast, is on looking at a wider variety of bugs across several benchmarks and proposing a meta-tool to examine the correlations.

Z-ranking [17] is a technique for ranking the output of static analysis tools so warnings that are more important will tend to be ranked more highly. As our results suggest, having such a facility in the tools we studied would be extremely useful. Z-ranking is intended to rank the output of a particular bug checker. In our paper, however, we look at correlating warnings across tools and across different checkers.

In general, since many of these Java bug finding tools have only been developed within the last few years, there has not been much work comparing them. One article on a developer web log by Jelliffe [15] briefly describes experience using JLint, FindBugs, PMD, and CheckStyle (a tool we did not study; it checks adherence to a coding style). In his opinion, JLint and FindBugs find different kinds of bugs, and both are very useful on existing code, while PMD and CheckStyle are more useful if you incorporate their rules into projects from the start.

8 Conclusion

We have examined the results of applying five bug-finding tools to a variety of Java programs. Although there is some overlap between the kinds of bugs found by the tools, mostly their warnings are distinct. Our experiments do, however, suggest that many tools reporting an unusual number of warnings for a class is correlated with a large breadth of unique warnings, and we propose a meta-tool to allow developers to identify these classes.

As we ran the tools and examined the output, there seemed to be a few things that would be beneficial in general. The main difficulty in using the tools is simply the quantity of output. In our opinion, the programmer should have the ability to add an annotation or a special comment into the code to suppress warnings that are false positives, even though this might lead to potential future problems (due to changes in assumptions). Such a mechanism seems necessary to help reduce the sheer output of the tools. In Section 6 we proposed adding this as a feature of the meta-tool.

In this paper we have focused on comparing the output of different tools. An interesting area of future work is to gather extensive information about the actual faults in programs, which would enable us to precisely identify false positives and false negatives. This information could be used to determine how accurately each tool predicts faults in our benchmarks. We could also test whether the two metrics we proposed for combining warnings from multiple tools are better or worse predictors of faults than the individual tools.

Finally, recall that all of the tools we used are in some ways unsound. Thus the absence of warnings from a tool does not imply the absence of errors. This is certainly a necessary tradeoff, because as we just argued, the number of warnings produced by a tool can be daunting and stand in the way of its use. As we saw in Section 3, without user annotations a tool like ESC/Java that is still unsound yet much closer to verification produces even more warnings than JLint, PMD, and FindBugs. Ultimately, we believe there is still a wide area of open research in understanding the right tradeoffs to make in bug finding tools.

Acknowledgments

We would like to thank David Cok and Joe Kiniry for helping us get ESC/Java 2 running. We would also like to thank Cyrille Artho for providing us with a beta version of JLint 3.0. Finally, we would like to thank Atif Memon, Bill Pugh, Mike Hicks, and the anonymous referees for their helpful comments on earlier versions of this paper.

References

[1] C. Artho. Finding faults in multi-threaded programs. Master's thesis, Institute of Computer Systems, Federal Institute of Technology, Zurich/Austin, 2001.

[2] D. Bacon, J. Block, J. Bogoda, C. Click, P. Haahr, D. Lea, T. May, J.-W. Maessen, J. D. Mitchell, K. Nilsen, B. Pugh, and E. G. Sirer. The "Double-Checked Locking is Broken" Declaration. `http://www.cs.umd.edu/˜pugh/java/memoryModel/DoubleCheckedLocking.html`.

[3] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M.-Y. Wong. Orthogonal defect classification - a concept for in-process measurements. *IEEE Transactions on Software Engineering*, 18(11):943–956, Nov. 1992.

[4] D. Cok. Personal communication, Apr. 2004.

[5] D. Cok and J. Kiniry. ESC/Java 2, Mar. 2004. `http://www.cs.kun.nl/sos/research/escjava/index.html`.

[6] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting Finite-state Models from Java Source Code. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 439–448, Limerick Ireland, June 2000.

[7] R. F. Crew. ASTLOG: A Language for Examining Abstract Syntax Trees. In *Proceedings of the Conference on Domain-Specific Languages*, Santa Barbara, California, Oct. 1997.

[8] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *ICSE 2000, Proceedings of the 22nd International Conference on Software Engineering*, pages 449–458, Limerick, Ireland, June 7–9, 2000.

[9] C. Flanagan and K. R. M. Leino. Houdini, an Annotation Assitant for ESC/Java. In J. N. Oliverira and P. Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods*, number 2021 in Lecture Notes in Computer Science, pages 500–517, Berlin, Germany, Mar. 2001. Springer-Verlag.

[10] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 234–245, Berlin, Germany, June 2002.

[11] K. Havelund and T. Pressburger. Model checking JAVA programs using JAVA pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[12] G. J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[13] D. Hovemeyer and W. Pugh. Finding Bugs Is Easy. `http://www.cs.umd.edu/˜pugh/java/bugs/docs/findbugsPaper.pdf`, 2003.

[14] IEEE. *IEEE Standard Classification for Software Anomalies*, Dec. 1993. IEEE Std 1044-1993.

[15] R. Jelliffe. Mini-review of Java Bug Finders. In *O'Reilly Developer Weblogs*. O'Reilly, Mar. 2004. `http://www.oreillynet.com/pub/wlg/4481`.

[16] JLint. `http://artho.com/jlint`.

[17] T. Kremenek and D. Engler. Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations. In R. Cousot, editor, *Static Analysis, 10th International Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 295–315, San Diego, CA, USA, June 2003. Springer-Verlag.

[18] PMD/Java. `http://pmd.sourceforge.net`.