Evaluating and Tuning a Static Analysis to Find Null Pointer Bugs[*]

David Hovemeyer, Jaime Spacco, and William Pugh
Dept. of Computer Science
University of Maryland
College Park, MD 20742 USA
{daveho,jspacco,pugh}@cs.umd.edu

ABSTRACT

Using static analysis to detect memory access errors, such as null pointer dereferences, is not a new problem. However, much of the previous work has used rather sophisticated analysis techniques in order to detect such errors.

In this paper we show that simple analysis techniques can be used to identify many such software defects, both in production code and in student code. In order to make our analysis both simple and effective, we use a non-standard analysis which is neither complete nor sound. However, we find that it is effective at finding an interesting class of software defects.

We describe the basic analysis we perform, as well as the additional errors we can detect using techniques such as annotations and inter-procedural analysis.

In studies of both production software and student projects, we find false positive rates of around 20% or less. In the student code base, we find that our static analysis techniques are able to pinpoint 50% to 80% of the defects leading to a null pointer exception at runtime.

Categories and Subject Descriptors

F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*program analysis*

General Terms

Experimentation, Measurement, Reliability

Keywords

Static analysis, testing

1. INTRODUCTION

Much recent research has investigated fairly sophisticated static analysis techniques for finding subtle bugs in programs. While subtle bugs certainly do exist, our experience has been that many interesting bugs arise from simple and common coding mistakes, and can be found using relatively simple analysis techniques. The challenge in developing an effective tool to find these bugs lies not so much in thinking of sophisticated analysis techniques as in identifying and evaluating simple analysis techniques which in combination are effective at finding the desired category of bugs.

In this paper, we describe a static analysis to find null pointer bugs in Java programs, and some of the simple analysis techniques we used to make it more accurate. We also discuss experiments we conducted to measure the effectiveness of the analysis, including a novel study of student programming projects. We find that the analysis finds many real bugs in analyzed programs, and the evaluation of the analysis on real software suggests several ways it could be extended to find even more real bugs.

1.1 Background

We developed our null-pointer analysis as part of FindBugs [8], which is an open source static bug-finding tool for Java. It works by analyzing Java class files at the bytecode level, and implements some of these bugs 100 common coding mistakes, including null pointer dereferences.

One of our goals in developing FindBugs is to help find common coding errors in student programming projects. In order to find out what kinds of mistakes students make in their programs, we developed Marmoset, an automated project snapshot, submission and testing system [10]. Like similar project submission systems, it allows students to submit versions of their projects to a central server, which automatically tests them and records the results. A novel feature of Marmoset is that it collects fine-grained code snapshots as students work on projects: each time a student saves her work, it is automatically committed to a CVS repository. The detailed development histories recorded by Marmoset combined with a suite of unit tests that provide fairly complete coverage of each project have yielded interesting and useful ways to validate the analyses implemented in FindBugs, as we will discuss later on in the paper.

2. NULL POINTER ANALYSIS

Exceptions due to dereferencing a null pointer are a very common type of error in Java programs. Some null pointer bugs arise because of null values loaded from the heap or

[*]Supported by NSF grant CCR-0098162 and by an IBM Eclipse Innovation award.

// org.eclipse.jdt.internal.ui.compare.JavaStructureDiffViewer

```
Control c= getControl();
if (c == null && c.isDisposed())
  return;
```

Figure 1: An obvious null pointer dereference in Eclipse 3.0.1

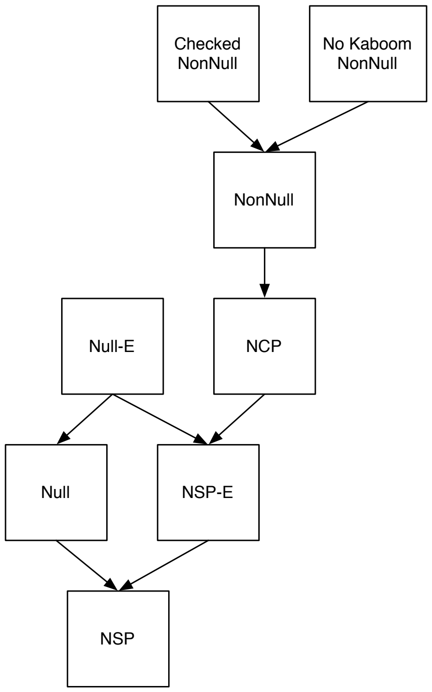

Figure 2: Dataflow lattice for null pointer analysis

passed from a distant call site, and might require sophisticated analysis techniques to find. However, our experience studying real Java applications and libraries has shown that many null pointer bugs are the result of simple mistakes, such as using the wrong boolean operator. An example of a simple null pointer bug is shown in Figure 1.

In this section, we describe the analysis used in FindBugs to determine where null pointer dereferences might occur. The design of this analysis illustrates some interesting examples of trade-offs needed to find real bugs without producing too many false warnings.

2.1 The Basic Analysis

The null pointer analysis is a forward intra-procedural dataflow analysis performed on a control-flow graph representation of a Java method. The dataflow values are Java stack frames containing "slots" representing method parameters, local variables, and stack operands. Each slot contains a single symbolic value indicating whether the value contained in the slot is definitely null, definitely not null, or possibly null. The lattice of symbolic values is shown in Figure 2, and the meaning of each value is described in Table 1.

```
boolean b;

if (p != null)
  b = true;
else
  b = false;

if (b) {
  p.f();
}
```

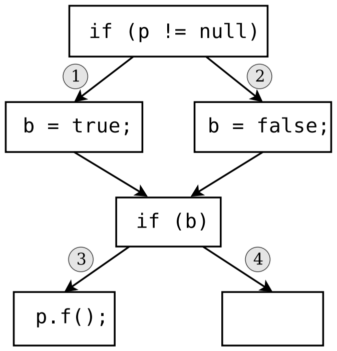

Figure 3: Example of an infeasible path. The dereference `p.f()` would cause a null pointer exception only on the path 2 → 3, which is infeasible.

2.1.1 Infeasible Paths

Because our null pointer analysis is meant to find bugs, it is important to have high confidence that a value really can be null at runtime before issuing a warning about a possible null pointer dereference. If too many false warnings are produced, the tool will not be worth the developer's time to use. Infeasible control paths are a common source of inaccuracy in dataflow analysis, and avoiding them is an important challenge in the design of an analysis to find null pointer bugs.

Often, a dereference of a possibly-null value v will be guarded by an explicit comparison of v to null, with the dereference occurring only when v is not null. In such cases, it is easy for the analysis to determine that the dereference of v is protected. A more difficult analysis problem arises when some other condition b guards the dereference of v. If b implies $v \neq$ null then the dereference of v is safe. An example of an indirect null check is shown in Figure 3. A naïve analysis might assume that `p` could be null at the call to `p.f()`, resulting in a false warning.

Unfortunately, determining which conditions entail a guarantee that a value is or is not null is a difficult problem, and is undecidable in general. In some cases, the condition checked may originate from outside the scope of the current analysis, such as a parameter passed into the method. Rather than trying to use a sophisticated analysis to untangle the meaning of conditions, we simply assume that *all* conditions are opaque null checks. The analysis implements this assumption as follows.

The meet of a definitely Null value with any other value results in an NSP value—Null on a Simple Path. At runtime, an NSP value is either null or non-null depending on the outcome of a single conditional branch statement. Assuming that the branch edge that causes the value to be null is feasible, dereferencing it could cause a null pointer exception, so the analysis issues a warning.

Whenever the analysis encounters a non-exception control split—in other words, an ordinary branch—it changes all NSP values to NCP, Null on a Complex Path. At runtime, an NCP value is either null or non-null depending on the outcome of *two or more* conditional branch statements. Because the conditions might be correlated, the analysis does not assume that it is possible for NCP values to be null at runtime.

		Warning if ...	
Value	Meaning	Dereferenced	Compared to null [†]
Null	Value definitely null	High	Medium
Null-E	Value null on exception path	Medium	Low
NonNull	Value definitely non-null	—	Low
Checked NonNull	Value non-null due to comparison	—	Medium
No Kaboom NonNull	Value non-null because it was dereferenced	—	High
NSP	Null on Simple Path	Medium	—
NSP-E	Null on Simple Path due to exception	Low	—
NCP	Null on Complex Path	—	—

[†]Low priority if the check does not create dead code

Table 1: Symbolic values used in the null pointer analysis

Statement	Value of `p`
`p = null`	Null
`p = this`	NonNull
`p = new ...`	NonNull
`p = "string"`	NonNull
`p = Foo.class`	NonNull
`p = q.x`	NCP
`p = a[i]`	NCP
`p = f()`	NCP

Table 2: Modeling statements in the null pointer analysis

Because no warnings are reported when NCP values are dereferenced, this technique misses some real bugs. However, it ensures that a very high percentage of warnings that are reported do correspond to real errors. In general, if it is possible to achieve full branch coverage in a method, then every dereference of a NSP is a real, exploitable bug. In the future we may augment the analysis to be smarter about inferring the effects of conditions which aren't direct null checks. (For one approach to efficient and precise analysis of program states implied by boolean conditions, see ESP [2].)

2.1.2 Modeling Values

On method entry, parameter values are assumed to be NCP, Null on a Complex Path. This is a conservative assumption reflecting the fact that the analysis has no *a priori* justification to assume that parameters are either null or non-null. An exception is the reference to the receiver object (`this`), which is set to NonNull in instance methods.

Statements are modeled as shown in Table 2. In this table, `p` refers to the value computed by an expression—in other words, the value that results from executing a statement. `q` refers to an arbitrary object reference, `a` refers to an arbitrary array reference, and `Foo` refers to an arbitrary class. In general, the analysis treats unknown values as NCP, Null on Complex Path.

2.1.3 Control Flow

Control joins are modeled in the usual way, by taking the meet in the lattice of the values in corresponding slots. Where possible, branches on comparisons to null are used to gain information about the tested value: we use the direction of the resulting branch to make the value either Null or NonNull on the appropriate control edge.

```
class Foo implements Cloneable {
  HashSet contents;
  ...

  public Object clone() {
    Foo dup = null;
    try {
      dup = super.clone();
    } catch (CloneNotSupportedException e) {
      // Can't happen
    }

    // Make deep copy of contents
    dup.contents = (HashSet)dup.contents.clone();

    return dup;
  }
}
```

Figure 4: An infeasible exception handler

Exception paths are handled specially. On entry to an exception handler, all Null values are replaced with Null-E, and all NSP values are replaced with NSP-E. We maintain this distinction because it is common for some call sites targeting methods declaring a checked exception to know that the checked exception cannot occur. A typical example involving an infeasible `CloneNotSupportedException` handler is shown in Figure 4. By keeping track of values which are only null on an exception path, we can lower the precedence of warnings emitted for dereferences of such values.

2.1.4 Redundant Comparisons

Interestingly, a significant number of null comparisons occurring in real programs are redundant, because the value compared to null is either definitely null or definitely non-null. Sometimes, this is because of defensive programming. A less benign cause of redundant comparisons is that a value was unconditionally dereferenced, and checked against null afterward. This often indicates a real error: if the value really can be null, the comparison should certainly be done before the dereference. (Xie and Engler [11] provide a general account of using redundant code to find bugs.)

FindBugs reports a high priority warning for all comparisons of No-Kaboom to null, since this strongly suggests that the value really can be null at the earlier location where it is dereferenced. Other redundant comparisons are considerably less likely to represent real bugs. We use a simple heuristic to determine which of the remaining cases is likely to be worth reporting: if the infeasible branch of a redundant comparison creates a nonempty region of dead code that does something other than unconditionally throw an exception, then we report a medium priority warning. Consider the following code fragment:

```
p = new Object();
q = new Object();
r = new Object();
...

if (p != null) { // defensive null check
  x = p.hashCode();
}

if (q == null) { // defensive null check
  throw new NullPointerException("q is null");
}

if (r != null) { // probable logic error
  y = r.hashCode();
} else {
  doSomethingElse();
}
```

All three null comparisons are redundant, since neither `p`, `q`, nor `r` can be null at runtime. The comparison `p != null` is defensive, because it does not cause any dead code. The comparison `q == null` does create dead code: however, the dead code simply throws an exception, so this case is also defensive programming. The comparison `r != null`, on the other hand, is more likely to indicate programmer confusion: the existence of executable code in the unreachable `else` block suggests a logic error.

In any case, it is important to ensure that the dead code resulting from redundant comparisons does not pollute the results of the analysis. Our analysis marks the frame on the infeasible branch of a redundant comparison as a special "Top" value, which serves as the meet identity element. This effectively makes the dead code invisible to the analysis. In the previous example, this would ensure that `p` and `q` retain the value NonNull, even after their respective null checks.

2.1.5 Assertions

Another form of infeasible control flow which must be considered in a null pointer analysis is exceptions due to failed assertions. Generally, an assertion in Java is a method that takes a boolean argument and throws an exception if the argument is false:

```
// throws exception if p null:
checkAssertion(p != null);
p.f(); // safe
```

We handle these kinds of assertions by simply changing any Null or NSP values to NCP following a call to a method containing the substring "abort", "assert", "check", "error", or "failed".

Another form of assertion is a method that throws an exception unconditionally:

```
if (p == null)
  error("p is null"); // throws exception
p.f(); // safe
```

We handle these cases using a simple inter-procedural analysis that identifies methods which throw an exception unconditionally, and at each call site, removing the control edge representing a normal return from those methods.

2.1.6 Finally Blocks

A `finally` block in Java is a region of code associated with a `try` statement which is guaranteed to be executed no matter how control leaves the `try` block. The Java source to bytecode compiler will emit code for a `finally` block either by duplicating it in the generated bytecode, or by emitting a `jsr` subroutine. Two issues arise regarding finally blocks.

The first issue is how to represent `jsr` subroutines in the control flow graph. For FindBugs, we decided to inline them into their call sites. This makes `jsr` and `ret` instructions used to call and return from `jsr` subroutines behave like `goto` instructions as far as the dataflow analysis is concerned. While this could theoretically result in an exponential increase in the size of the resulting control flow graph, we have not observed this in practice.

The second issue is how to handle warnings for code inside `finally` blocks. For most kinds of warnings, including null pointer dereferences, as long as the warning describes a bug feasible for at least one expansion of the block, the warning is valid. However, redundant comparison warnings are only valid if the comparison is redundant for *every* expansion, and is always redundant for the same reason. We use the method source line number table to keep track of duplicated code, and only emit redundant comparison warnings if all redundant comparisons for a particular line are in agreement.

2.2 Extending the Basic Analysis

In this section, we discuss some inter-procedural extensions to our basic null pointer analysis.

2.2.1 Unconditionally Dereferenced Parameters

One common source of null pointer errors we have observed in real programs is confusion about whether method parameters may be null. A sure sign that the programmer believes a parameter should be non-null is that it is dereferenced unconditionally. If a caller ever passes a null value for such a parameter, a null pointer exception is guaranteed.

To find methods which unconditionally dereference a parameter, we perform a backward dataflow analysis to compute the set of parameter values guaranteed to be dereferenced on all forward paths. The set at the entry of the method's control flow graph is the set of unconditionally dereferenced parameters. The analysis excludes runtime exception control edges from consideration, unless they are thrown via an explicit `throw` statement. To see why it is necessary to exclude "implicit" runtime exceptions, consider the following method:

```
boolean sameHashCode(Object p, Object q) {
  return p.hashCode() == q.hashCode();
}
```

Although it is possible for the call to `p.hashCode()` to throw a null pointer exception which bypasses the call to `q.hashCode()`, that behavior is not part of the expected behavior of the

Annotation	Context	How checked
@NonNull	Parameter	Callers must not pass Null or NSP
	Return value	Method must not return Null or NSP
@CheckForNull	Parameter	Method must not dereference unconditionally
	Return value	Callers must not dereference unconditionally

Table 3: Supported annotations for method parameters and return values

method. So, our analysis would consider *both* parameters of this method to be dereferenced unconditionally.

Once we have computed the set of methods which unconditionally dereference a parameter, we examine all call sites to find places where a possibly-null value (Null or NSP in the lattice) is passed to a method which might unconditionally dereference it. An obvious difficulty in Java is determining which methods might be called at polymorphic call sites. Our analysis conservatively assumes that unless the type of the receiver object is known exactly it could be any concrete subtype. Some methods, such as `Object.equals()`, have a large number of potential target methods, and could therefore generate a large number of false warnings. To reduce the effect of these false positives, we issue a higher priority warning for cases where there is only one known target, or when all known targets dereference the parameter unconditionally.

2.2.2 Parameter and Return Value Annotations

A recurring issue in program analysis to find bugs is trying to deduce the *expected* behavior of some piece of code. In null pointer analysis, it is often unclear where the blame lies when an inconsistency is detected. For example, when an unconditional parameter dereference results in a null pointer exception, should we blame the caller or the callee?

Specifications are a simple way for the programmer to make the job of the analysis easier by explicitly marking which values must not be null and which may be null. (Another tool that leverages lightweight annotations to find memory bugs, including null pointer dereferences, is LCLint [5]). Our analysis can use two kinds of specifications on method parameters and return values: @NonNull, which indicates that a value must not be null, and @CheckForNull, which indicates that a value might be null. These specifications are conveyed using Java source annotations. Table 3 lists a summary of how these annotations are checked.

Note the name @CheckForNull is carefully chosen, and it not simply the converse of @NonNull (which might, perhaps, be annotated @Nullable). The standard Java semantics, and the default meaning associated with a reference parameter or return value, is that it could be null. However, in many cases the caller can easily determine from context, without an explicit null-check, that the value is nonnull. For example, a get method invoked on a List could return a null value if the List contained null, but in many/most situations, the programmer may know that the List will not contain null values, and requiring an explicit null check would just obfuscate the code and annoy the developer. The annotation @CheckForNull is intended for cases where good programming practice recommends that the value be checked for null. For example, the return value from BufferredReader.readLine() or the argument to equals(Object) should be checked for null, and could be marked @CheckForNull. Other methods are more of a judgment call. For example, the get method for Map returns null to signal no entry for a key. But in many use cases, you know that the key is contained in the Map, and requiring an explicit null check would be unproductive.

@NonNull and @CheckForNull annotations are inherited by subclasses. Annotations on method parameters may be relaxed by subclasses but not strengthened (e.g., changing a @NonNull annotation to @CheckForNull on a parameter), while annotations on return types may be strengthened but not relaxed (e.g., changing a @CheckForNull annotation on a return type to @NonNull). In this way, annotations behave similar to covariant and contravariant constraints on method parameters and return types.

Because annotations are a contract applying to *all* subclasses, they do not suffer imprecision due to the difficulty of resolving target methods at polymorphic call sites. However, they do impose an additional restriction on method callers not to pass a null value or unconditionally dereference a return value, even when they know that to be safe in the calling context.

3. EXPERIMENTAL RESULTS

In this section we present results and observations from running our null-pointer analysis over both student code and production code. The analysis of student projects was performed with FindBugs version 0.9.2. The analysis of production code was performed with FindBugs version 0.9.3, in order to take advantage of updated heuristics for redundant null comparison warnings.

3.1 Student Code

Using Marmoset, we have collected snapshots of students' source code for the second semester object oriented programming course at the University of Maryland. We focused our analysis on two projects, a Binary Search Tree (BST) for which we recorded 3,357 unique compilable snapshots from 73 students and a WebSpider for which we recorded 3,771 unique compilable snapshots from 92 students (the projects were from different semesters; hence the different number of students).

The student project repositories collected by Marmoset have several nice properties: First, student repositories are easy to configure and run. Second, each project has unit tests that provide fairly complete coverage of major project features. Finally, student repositories represent several dozen attempts to implement the same project by programmers with varying abilities, experience, and coding styles.

We would like to determine both what null pointer exceptions that arise during unit testing correspond to defects identified by our static analysis, and what warnings generated by our static analysis correspond to actual software defects. Null pointer exceptions not predicted by any static analysis warning are *false negatives*, and warnings not corresponding to a feasible null pointer exception are *false posi-*

Project	Snapshots with NPE	With warning	Observed false neg. %
Search Tree	71	38	46
Web Spider	162	127	21

Project	Warnings	With NPE	Observed false pos. %
Search Tree	40	36	10
Web Spider	129	101	21

Table 4: Observed false negative and false positive rates for null pointer warnings in student projects (with annotations)

Project	Snapshots with NPE	With warning	Observed false neg. %
Search Tree	71	1	98
Web Spider	162	47	70

Project	Warnings	With NPE	Observed false pos. %
Search Tree	2	2	0
Web Spider	77	75	2

Table 5: Observed false negative and false positive rates for null pointer warnings in student projects (without annotations)

tives. The goal of any static analysis to find bugs is obviously to make the false negative and false positive rates as low as possible.

The main question in measuring false negatives and false positives is which snapshots and warnings to count. There are two issues to consider in answering this question. First, we cannot easily evaluate the accuracy of a warning if the statement it identifies is never executed. To resolve this issue, we collected code coverage data for each unit test, and excluded all warnings never covered by any test case from consideration. The second issue is slightly more subtle. Because Marmoset collects fine-grained snapshots, it tends to record a single warning or runtime exception many times. If we consider all snapshots and warnings, we tend to overcount those that persist through many successive versions. This will over count warnings that are false positives, and may undercount exceptions that are false negatives. To solve this problem, we only counted warnings that are either removed in the subsequent snapshot, or are present in the final snapshot, as potential false positives. Likewise, for potential false negatives we only counted the first in a chronological series of snapshots where a particular null pointer exception occurred.

After selecting the subset of snapshots and warnings to count, we calculated false negatives and false positives as follows:

$$\text{False negatives} = \frac{\text{Snapshots w/ exception but no warning}}{\text{Snapshots with exception}}$$

$$\text{False positives} = \frac{\text{Warnings w/ no exception thrown}}{\text{Warnings}}$$

Tables 4 and 5 show the results for the two projects we studied. With annotations on selected method parameters and return values, the analysis was able to find between 50% and 80% of all null pointer bugs, with a low false positive rate. Without annotations, the false positive rate drops to near zero, but the false negative rate is much worse, finding only 30% of the null pointer bugs for the Web Spider project, and only 2% for the Search Tree project. From this result we draw two conclusions. First, strictly intra-procedural analysis is not sufficient to find most null pointer bugs. Second, annotations on method parameters and return values can effectively extend an intra-procedural analysis to find a significant number of null pointer bugs that would otherwise require inter-procedural analysis. Because the student projects were relatively small and not very complex, we cannot confidentially predict that the same false negative rates would hold for large production applications. However, we believe that annotations could find a significant chunk of the null pointer bugs in such applications.

Warning type	Serious	False	% Serious
Null dereference	73	16	82
No Kaboom RCN comparison	33	15	69
Other RCN	15	17	47

Table 6: Serious bugs and false warnings in Eclipse 3.0.1 for null dereference and redundant comparison to null (RCN) warnings

3.2 Production Code

We also evaluated the null pointer checker on one production application: Eclipse 3.0.1 ([3]). While student programming projects are a useful testbed, they may not represent the kind of code that professional programmers write. Our goal in evaluating the analysis on production software was to count how many real bugs the analysis found, and also measure the false warning rate. The results of this evaluation are shown in Table 6. We obtained this data by manually classifying each null-pointer and redundant null comparison warning by hand as either a serious bug or a false warning. This process involves fallible human judgment; however, we tried to err on the side of only marking a warning as a real bug if we were confident it would be worth fixing.

In general, the null dereference warnings (which correspond to dereferences of Null and NSP values) were very accurate. The most common source of inaccurate warnings was falling off the end of a chain of `if/else` statements, or falling through the default case of a switch: the analysis assumes both of these cases represent feasible control flow, although in many cases these control edges are not feasible. The redundant null comparison warnings involving a previously dereferenced (No Kaboom) value found some genuine bugs. With the use of the dead-code heuristic, the other RCN warnings also identified 15 real bugs, with a false positive rate only slightly higher than 50%.

We have not yet evaluated the inter-procedural or annotation-based versions of the analysis on production code. However, the basic intra-procedural analysis finds a fairly large number of real bugs on its own. Our preliminary experience is that performing this kind of analysis across method boundaries is very sensitive to calling context: it is typical for a single call site to exercise only a small number of paths through the called method or methods. We believe that by concentrating on the unambiguous cases, such as passing an NSP value for a parameter which is unconditionally dereferenced by the called method, we will be able to find some additional bugs without too many false positives.

4. RELATED WORK

Many static analysis tools for finding memory errors have been developed: for example, LCLint [5], PREfix [1], and Metal [4] are good examples. Because they analyze C programs, some of the bugs they target are more low level than those occurring in Java. LCLint uses specifications to improve precision in the absence of extensive inter-procedural analysis; our use of annotations is meant to address the same problem.

ESC/Java [7] is a static checker for Java that uses annotations, such as null and non-null, in an effort to prove useful properties about a program. While FindBugs' null pointer analysis can also benefit from annotations, there are key differences between the two tools. First, ESC/Java (as well as many other specification-based approaches to static checking) tries to find *all* violations of a null or non-null annotation, which can find more real bugs at the cost of a (potentially much) higher false positive rate.

In [6], Fähndrich and Leino discuss many important issues involved in retrofitting an existing type-safe object-oriented language with augmented types containing information about the potential nullity of a reference. Although their work focuses on C#, most of the issues they cover apply to Java as well. The primary difference between this work and our work is in scope. They use annotations as a tool to attack a larger, more general problem, while we use annotations as a convenient lightweight mechanism to improve the precision of our bug checker. These two goals are complementary in that FindBugs could greatly benefit from the information provided by a new language feature such as the non-null types proposed by Fähndrich and Leino.

In [9], Rountev et. al. propose a methodology for precisely determining the imprecision of applying conservative static analysis to a software artifact. The technique involves using dynamic analysis (executing tests) to determine a feasible set of analysis facts, and then using proofs of infeasibility to identify the areas of imprecision left over. Our goals and methodology are at least somewhat similar, in that we focus on identifying the intersection of static and dynamic analysis results with the aim of making the intersection as large as possible.

5. REFERENCES

[1] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Software—Practice and Experience*, 30:775–802, 2000.

[2] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 57–68. ACM Press, 2002.

[3] Eclipse. `http://www.eclipse.org`, 2005.

[4] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation, San Diego, CA*, October 2000.

[5] D. Evans. Static detection of dynamic memory errors. In *Proceedings of the SIGPLAN Conference on Programming Languages, Design, and Implementation*, 1996.

[6] Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 302–312, New York, NY, USA, 2003. ACM Press.

[7] C. Flanagan, K. Leino, M. Lillibridge, C. Nelson, J. Saxe, and R. Stata. Extended static checking for Java, 2002.

[8] David Hovemeyer and William Pugh. Finding bugs is easy. In *Companion of the 19th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Vancouver, BC, October 2004.

[9] Atanas Rountev, Scott Kagan, and Michael Gibas. Evaluating the imprecision of static analysis. In *PASTE '04: Proceedings of the ACM-SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 14–16, Washington DC, USA, 2004.

[10] Jaime Spacco, Jaymie Strecker, David Hovemeyer, and William Pugh. Software repository mining with Marmoset: An automated programming project snapshot and testing system. In *Proceedings of the Mining Software Repositories Workshop (MSR 2005)*, St. Louis, Missouri, USA, May 2005.

[11] Yichen Xie and Dawson Engler. Using redundancies to find errors. In *SIGSOFT '02/FSE-10: Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 51–60, Charleston, South Carolina, USA, 2002.