XFindBugs: eXtended FindBugs for AspectJ

Haihao Shen, Sai Zhang, Jianjun Zhao, Jianhong Fang, Shiyuan Yao
School of Software, Shanghai Jiao Tong University
800 Dongchuan Road, Shanghai 200240, China
{haihaoshen, saizhang, zhao-jj, fjh, ysyadam}@sjtu.edu.cn

ABSTRACT
Aspect-oriented software development (AOSD) is gaining popularity with the wider adoption of languages such as AspectJ. However, though the state-of-the-art aspect-oriented programming environment (such as AJDT in the Eclipse IDE) provides powerful capability to check the syntactic or grammar errors in AspectJ programs, it fails to detect potential semantic defects in aspect-oriented software systems. In this paper, we present XFindBugs, an eXtended FindBugs for AspectJ, to help programmers find potential bugs in AspectJ applications through static analysis. XFindBugs supports 17 bug patterns to cover common error-prone features in an aspect-oriented system, and integrates the corresponding bug detectors into the FindBugs framework. We evaluate XFindBugs on a number of large-scale open source AspectJ projects (306,800 LOC in total) and find 257 previously unknown defects. Our experiment also indicates that the bug patterns supported in XFindBugs exist in real-world softwares, even for mature applications by experienced programmers.

1. INTRODUCTION
Static analysis for software defect detection is a promising technique to improve software quality. Because of the sheer complexity of modern programming languages, the potential for misuse of language features, API rules or simply bad programming practice may be enormous. Static analysis techniques can explore abstractions of all possible program behaviors, and thus are not limited by the quality of test cases in order to be effective. Static analysis tools, such as [8,10,11,19,22], serve an important role in raising the awareness of developers about subtle correctness issues. In addition to finding existing bugs, these tools can also help programmers to prevent future defects. FindBugs [8], one of the most popular static analysis tools, is becoming widely used in Java community. FindBugs implements a set of bug detectors for a variety of common bug patterns (code idioms that are likely to be errors [25]), and uses them to find a significant number of bugs in real-world applications and libraries [14,15].

Aspect-Oriented Programming (AOP) [27] has been proposed as a technique for improving separation of concerns in software design and implementation. It is gaining popularity with the wider adoption of languages such as AspectJ. AspectJ is a seamless extension of Java. An AspectJ program can be divided into two parts: base code which includes classes, interfaces, and other language constructs as in Java, and aspect code which includes aspects for modeling crosscutting concerns in the program. However, though the state-of-the-art aspect-oriented programming environments (such as AJDT [5] in the eclipse [7] IDE) provide powerful capability to check the syntactic or grammar errors in AspectJ programs, they fail to detect potential semantic defects in software systems. For example, the type checker used in AJDT only checks the syntactic correctness of the program, but fails to identify or even to generate a warning about the type conflicts introduced by an aspect. In such cases, once the class containing an introduced field with a conflicting type is instantiated, the whole program will be terminated abruptly.

Although the executable code of an AspectJ program is pure Java bytecode, the existing bug patterns and defect detection tools for Java bytecode might not be applied directly. In addition to the bytecode that corresponds to the source code (e.g., to bodies of advices), the compiled bytecode of an AspectJ program contains extra code inserted by the compiler during the weaving process. After weaving, the source code level aspect, advice or intertype declaration information has been translated into pure Java bytecode instructions, and therefore is no longer preserved. In fact, in our experimental study, none of the bugs found by XFindBugs presented in Section 5 can be detected directly by FindBugs.

In this paper, we present XFindBugs, an eXtended FindBugs for AspectJ. XFindBugs defines a catalog of 17 bug patterns for aspect-oriented features, and implements a set of bug detectors on top of the FindBugs analysis framework. Bug patterns abstract common misunderstandings of language features, API rules and bad programming practice. They help programmers get a better understanding of how to write bug-free code. We perform an empirical evaluation of XFindBugs on several AspectJ benchmarks and third-party large-scale applications (like GlassBox [9], AJHotDraw [1], and AJHSQDB [21]). XFindBugs finds 257 previously unknown defects in these subjects, some of which may even result in a software crash. The experiment also indicates that the bug patterns XFindBugs supports exist in real-world softwares, even in mature AspectJ applications by experienced programmers.

In summary, the main contributions of this paper are: (1) a systematic catalog of bug patterns for AspectJ programs, (2) design and implementation of XFindBugs, a static defect detection tool for AspectJ software, and (3) an empirical evaluation of XFindBugs on over 300KLOC, which evidences the practical issues.

The rest of the paper is organized as follows. We start to briefly introduce the background of FindBugs and discuss the error-prone features of AspectJ programs in Section 2. Section 3 presents the a catalog of bug patterns for AspectJ in detail. Section 4 summarizes the implementation issues on XFindBugs. Section 5 reports an empirical evaluation on XFindBugs. Related work and concluding remarks are given in Section 6 and Section 7, respectively.

2. BACKGROUND

We next briefly introduce the background of FindBugs and error-prone features in aspect-oriented programs.

2.1 FindBugs
FindBugs [8], an open-source static analysis tool, has been widely used for detecting programming defects in Java community. It pro-
vides an extensible plugin architecture in which bug detectors can be easily defined, each of which may correspond to several different bug patterns. Until now, there are more than 300 bug patterns supported by FindBugs. The detectors implemented in FindBugs use a variety of techniques. Many simple detectors use a visitor pattern over the classfile, often using a state machine to reason about values stored on the stack or in local variables. Some detectors also incorporate control flow and data flow information into analysis, for finding sophisticated defects like Null Pointer Bugs [26]. FindBugs has been evaluated on a number of commercial and open source projects. For example, FindBugs analyzed all 89 publicly available builds of JDK (from builds b12 to b105) [14] and generated over 370 warnings with high/medium priority. Google has also incorporated static analysis into its software development process [14, 30]. The developers in Google run FindBugs on Google’s Java code base, manually evaluated warnings, and filled bug reports as deemed appropriate. As a result, there are totally 1127 warnings reported by FindBugs in Google with medium/high priority. In both experiences, the help of FindBugs facilitates programmers to find the potential defects and then modify them quickly.

2.2 Error-prone Features in AspectJ Programs

AspectJ [3], a seamless aspect-oriented extension to the Java programming language, encapsulates crosscutting concerns for better modularity using constructs like pointcut, advice, and intertype declarations. These new aspectual features ease separation of concerns in software design and implementation, but also introduce new error-prone features to the traditional Java programs. For example, the join point model in AspectJ is defined in a lexical-level, and the selection relies on naming conventions. It is easy for programmers to pick up incorrect join points, particularly when using wildcards in pointcut designators (such as the bug patterns of the Pointcut category in Table 1). Also, more than one advice can be activated at the same join point, in which the advice invocation sequence would affect the program execution and one advice’s behavior may be altered by another advice (such as the bug patterns of the Advice category in Table 1). Furthermore, the intertype declaration mechanism (also called introduction or structural superimpositions [18]) offered by AspectJ can easily add new class members to override or extend the ones defined in the original class hierarchy dramatically (such as the bug patterns of the Introduction category in Table 1).

The error-prone features of AspectJ may easily lead to potential defects in an aspect-oriented system, even for experienced programmers or in mature applications (Section 5). However, the current state-of-the-art programming environment for AspectJ (such as AJDT [5]) fails to detect such defects or even fails to generate any warnings. Therefore, in order to handle the unique aspectual categories (Advice, Pointcut, and Introduction) according to their root causes. We also define a priority (Low, Medium, and High) for each bug pattern to indicate the severity degree. Since most bug patterns have a number of representation variant and alternatives, we choose the one that appears to be the most generally applicable. We next select six typical bug patterns in Table 1 to explain the symptom, cause, and cure. Explanations of other bug patterns can be found in an extended technical report [31].

3.1 Access Field Before Object Initialization

Accessing an object before its initialization will result in a NullPointerException at runtime. The AspectJ language provides a mechanism (the execution(ClassName.new()) pointcut) to intercept constructor calls. The advice can also take over the control flow of object initialization in the program, and leave the object uninitialized. For example, Programmers sometimes incorrectly define an advice like in Figure 1. In this example, accessing caller object through this(a) will cause a Null Pointer deference. The detector in XFindBugs looks for instructions in a classfile, and generates a warning where an object might be accessed before its initialization.

An example (found by XFindBugs) of this bug pattern is shown in Figure 1. The before advice accesses object a before its initialization, and the dereference of a in statement if(!a.s.equals("some value") will lead to a NullPointerException at runtime. As later described in Section 5, we are surprised to find that a similar bug instance could find its way into a mature, well-tested application.

```java
public class A {
    public String s = "Initialize s!";
}
public aspect B {
    pointcut beforeInitialize(A a):
        execution(A.new()) && this(a);
    before(A a): beforeInitialize(a) {
        if(!a.s.equals("some value")) {...}
}
```

Figure 1: Example of Access Field Before Object Initialization

We recommend programmers avoid accessing any uninitialized field in the before advice, or use defensive programming (as shown in Figure 2) to check the nullness when using fields, especially when the advice intercepts constructor calls.

3.2 Mismatching Of After Returning

There are two special cases of advice in AspectJ, after returning and after throwing, corresponding to the two ways a sub-
computation can return through a join point. Since the AspectJ compiler is unaware of the return type exposed by the after returning advice, when an after returning advice matches a void method, the return type will be treated as NULL. A bug example found by XFindBugs is shown in Figure 3. The advice if(a.s != null) {...} before(A a): beforeInitialize(a) {
    System.out.println("The return type is void!");
}

Figure 2: Corrected Example of AJAFBI

3.3 Singleton Aspect

In AspectJ, the default behavior of a non-abstract aspect is to have a single instance [6], and advice runs in the context of this instance. The aspect declaration also accepts a modifier, called "of" that provides other kinds of aspect instance behavior. This aspect instance initialization mechanism poses extra complexity and error-proneness to the programs. In case of the careless pointcut definition, when an aspect has an initialization pointcut which accidentally matches the self-initialization join point, the program will terminate with a runtime exception. This bug pattern, though it seems tricky or even ironic, is also found in our experiment.

We take a code snippet found by XFindBugs as an example (Figure 4). The pointcut initPoint() unintenionally matches the initialization of aspect B, and causes the program to terminate when the after() initPoint() advice body is executed. XFindBugs detects such anomaly in the AspectJ programs, and generates a warning, informing programmers should add a perthis keyword in the aspect declaration (Figure 5).

Figure 3: Example of Mismatching Of After Returning

before(A a): beforeInitialize(a) {
    System.out.println("The return type is void!");
}

Figure 5: Corrected Example of AJSA

Figure 4: Example of Singleton Aspect

3.4 The Return Of Proceed

The around advice in AspectJ has the special usage of selectively preempting the normal computation at the join point. Around advice runs in place of the join point it operates over, rather than before or after it, while the proceed form takes as arguments the context exposed by the around’s pointcut, and returns whatever the around is declared to return. However, within the body of an around advice, calling proceed() will invoke the next most specific piece of around advice (if there is any), which may lead to unexpectedly problems.

An example of The Return of Proceed found by the detector is shown in Figure 6. The program initializes the field Integer i in the interface I with value new Integer(3), after executing statement Object result = proceed(val). However, after the Integer around(int val) advice, field i has been assigned twice (with the same value new Integer(3)). This violates the Java language specification\(^2\), and a ClassFormatError will be thrown at runtime.

XFindBugs detects such defect and generates a warning for programmers, informing the potential conflicts (the field i in the interface I, and the proceed(val) statement in this example).

Figure 6: Example of The Return Of Proceed

3.5 The Negated Pointcut

As introduced in Section 2.2, the join point model in AspectJ is defined in a lexical-level, and the selection relies on naming conventions. So, when using wildcards specified in a pointcut to match join points, it is easy for programmers to pick up an incorrect join point. The negated pointcut here refers to those pointcuts like negatenoPoint(): "execution(* A.printString())", which use "*" to exclude certain join points during execution. The use of negated pointcut may ease to express certain complex pointcut expressions, but it also has serious side-effects which may even result in a program crash.

Consider the negated pointcut example in Figure 7, which will throw an ExceptionInitialzeError at runtime. The negatenoPointcut() is likely to intercept a number of unexpected join points, such as object construction or the main method. These unexpected intercepted join points will lead to an Object Access Before Initialization or Infinite Loop bug pattern.

3.6 Unchecked Intertype Declarations

The intertype declaration mechanism of AspectJ can introduce a new class member, to change the program structure statically. However, the type checker of the AspectJ compiler does not verify the type correctness of intertype declarations, which may result in program ambiguity [21], introduction conflicts or even runtime errors. The bug pattern Unchecked Intertype Declarations refers to

\(^2\)Public fields in a Java interface all have "public static final" modifiers after being compiled, and should not be assigned twice.
public class A {
    public void printString() {...}
    public static void main(String[] args) {
        new A().printString();
    }
}

public aspect B {
    pointcut negatepoint(): !execution(* A.subString());
    before(): negatepoint() {...}
}

Figure 7: Example of The Negated Pointcut

public class A {
    public static void main(String[] args) {
        new A().printString();
    }
}

public aspect B {
    public int A.x = "hello";
}

Figure 8: Example of Unchecked Intertype Declarations

4. IMPLEMENTATION ISSUES

To investigate the applicability of our proposed bug patterns, we implemented XFindBugs, an eXtended FindBugs for AspectJ programs. XFindBugs is built on top of the FindBugs analysis framework and supports the AspectJ compiler ajc v1.5 compiler. However, the program will terminate with a VerifyError as soon as it is executed.

Our XFindBugs implements the bug detector to check the type consistency of each intertype declaration, and generates corresponding warnings if any type rules in the Java language specification are violated.

public class A {
    public static void main(String[] args) {
        new A();
    }
}

public aspect B {
    public int A.x = "hello";
}

Figure 9: Bytecode Signature of Singleton Aspect Bug Pattern

• Do the bug patterns defined in this paper exist in real-world AspectJ applications?
• Can the tool XFindBugs find real potential defects?
• Can XFindBugs scale to large applications, or is there a real necessity for the usage of our tool?

5.2 Subject Programs

The subject programs used in this paper are collected from various sources. AJHotdraw [1], AHSQlDB [2], and GlassBox [9] are three mature large-scale AspectJ applications. The ajc and abc benchmarks are obtained from the ajc and abc compiler distribution packages. The design patterns program suite, which implements 23 design patterns, is described in [20]. The subject programs used in this evaluation are also widely used by other researchers to evaluate their work [12,32,33]. Table 2 shows the name of each subject program (Name), the number of lines of code in total (LOC), the number of advices (#Advice), the number of pointcuts (#Pointcut), and the number of intertype declarations (#Introduction). Several subject programs have multiple releases, we run XFindBugs on each available version.

Table 2: Subject Programs

<table>
<thead>
<tr>
<th>Name</th>
<th>LOC</th>
<th>#Advice</th>
<th>#Pointcut</th>
<th>#Introduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>AJHotdraw</td>
<td>38546</td>
<td>48</td>
<td>13</td>
<td>54</td>
</tr>
<tr>
<td>AHSQlDB</td>
<td>12566</td>
<td>30</td>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>GlassBox</td>
<td>39220</td>
<td>132</td>
<td>183</td>
<td>44</td>
</tr>
<tr>
<td>abc Benchmarks</td>
<td>4656</td>
<td>44</td>
<td>30</td>
<td>27</td>
</tr>
<tr>
<td>ajc Benchmarks</td>
<td>80936</td>
<td>34</td>
<td>24</td>
<td>43</td>
</tr>
<tr>
<td>Design Patterns</td>
<td>10821</td>
<td>15</td>
<td>24</td>
<td>43</td>
</tr>
</tbody>
</table>

5.3 Experiment Procedures

We conducted two different evaluations on XFindBugs to show XFindBugs can effectively detect potential defects in AspectJ programs.

• we extract the existing bug reports from AspectJ Bugzilla [4] and run XFindBugs on the reported buggy code, to see whether it can detect them.
• we run XFindBugs on the subject programs listed in Table 2, to see whether XFindBugs can find any previously unknown defects.

XFindBugs performs fully automated analysis and generates warnings for each suspicious code snippet. For each warning produced by XFindBugs, we also check the corresponding source code manually to confirm its validity.

5.4 Threats to Validity

Like any empirical evaluation, there are some threats to validity which must be taken into consideration in our experiment. Though XFindBugs has been evaluated on over 300KLOC Java and AspectJ code, the programs we investigated may be not representative enough. Therefore, we cannot claim XFindBugs can find every bug pattern instance in an arbitrary AspectJ application. Another
internal threat to validity is the false positives XFindBugs generates (discussed later in this section), for this reason, we also can not claim all warnings generated by XFindBugs are accurate.

Since the bytecode generated after the advice weaving process is compiler-specific, different AspectJ compilers may produce different bytecode. For this reason, though the current implementation of XFindBugs fully supports the AspectJ compiler ajc v1.5, one of the most widely used and well supported compilers, we can not claim XFindBugs can find bug patterns in the woven bytecode produced by a different AspectJ compiler.

5.5 Result and Analysis

5.5.1 Defects from AspectJ Bugzilla

We evaluated XFindBugs to check whether it can confirm known defects described in the literature. We picked six defects3 in the AspectJ Bugzilla. These real defects are reported by AspectJ programmers around the world during their own development. Six bug patterns, namely Assign Of Field Of SuperClass, The Return Of Proceed, The Negated Pointcut, Invoke Generic Type Method Indirectly, Useless Introduced Field, and The Different Access Modifier, are covered in the selected bug reports. As a result, XFindBugs confirms all six bug patterns by pinpointing the faulty code correctly.

5.5.2 Defects in Subject Programs

We used XFindBugs to detect potential bugs in a number of real-world AspectJ applications in Table 2. Table 3 shows the empirical results produced by XFindBugs. Column 1 shows the evaluated subject program’s name, columns 2 - 7 show the number4 of each bug pattern (Table 1) found by XFindBugs, column 8 (#All) summarizes the total bug pattern number, and column 9 (%FP) shows the rate of false positives after manually inspecting the source code.

From the table, we observe that XFindBugs finds totally 280 previously unknown defects, in which only 23 of them are false positives. In addition to six bug patterns from AspectJ Bugzilla, we demonstrate that a large part (12 out of 17) of bug patterns does exist in real-world applications and XFindBugs can scale well to find them within an acceptable false positive rate.

AJHotDraw. AJHotDraw is an aspect-oriented refactoring of JHotDraw, which is a well-designed open source Java framework for technical and structured 2D graphics. However, XFindBugs still finds two defects on its latest version. These two defects belong to the Misuse of Target bug pattern. We show the code sample taken from AJHotDraw in Figure 10 and 11.

84: void around(DrawingView drawingView) {
85: callCommandFigureSelectionChanged(drawingView) {
86: AbstractCommand command = (AbstractCommand)thisJoinPoint.getTarget();
87: proceed(drawingView);
88: }

Figure 10: Misuse Of Target in AJHotdraw: Line 84 – 90 in org.jhotdraw.econcerns.commands.UndoableCommand.

The two advices shown in Figure 10 and 11 both advise static methods in AJHotdraw. In this example, the thisJoinPoint.getTarget() would return NULL, and a NullPointerException will be thrown when dereferencing the return object.

AJHSQLDB. Like AJHotDraw, AJHSQLDB is another aspect-oriented refactoring case study on HSQLDB. It has more than 120 KLOC in the investigated version. XFindBugs also finds several unknown defects. We show the sample code of bug pattern Access Field Before Object Initialization in Figure 12. In this piece of code, the advice in line 72 intercepts the exception handling block in line 35. That is, if an exception is throw from line 35, the advice in line 72 will take the control flow. However, the variable pw used in line 77 still remains uninitialized (it is intended to initialized in line 43), and a NullPointerException will be thrown.

34: public ExceptionHandlingAbstractAspect() {
35: try {
36: catch (Exception e) {
37: if (LOG_CATCH_BLOCKS) {
38: pw = new PrintWriter(fw);
39: }
40: }
41: }
42: try {
43: pw = new PrintWriter(fw);
44: }
45: }
72: before (Exception e) : exceptionCatchBlocks(e)
76: if (thisJoinPointStaticPart && LOG_THROW_EXCEPTION) {
77: pw.println(thisJoinPointStaticPart)
78: ...
81: }

Figure 12: Access Field Before Object Initialization: Line 76 – 80 in org.hsqldb.aspect.ExceptionHandlingAbstractAspect.

In AJHSQLDB, another three Misuse Of Target bug pattern instances come from line 2619 to 2627, line 2646 to 2646, and line 2676 to 2685 in the org.hsqldb.util.DatabaseManagerSwing.aj file. Due to the space limitation, we do not show the code here.

Glassbox. Glassbox is a widely-used troubleshooting agent for Java applications that automatically diagnoses common problems. It is well tested and the latest version available is 2.0GA. XFindBugs finds one defect, belonging to the Scope Of Advice bug pattern in it. We show the sample code taken in Figure 13.

In Figure 13, the advice parameter sql has been modified within advice before (Statement, String sql) : topLevelDynamicSqlExec(statement, sql). However, like method parameters, advice parameters are local to the advice. In other words, reassigning sql in the above example will have no effect outside the advice. Therefore, XFindBugs treats it as a potential defect and generates a warning with priority Low.

182: before(Statement statement, String sql) : topLevelDynamicSqlExec(statement, sql) {
183: if (sql==null) {
184: sql = "NULL dynamic sql statement";
185: }
186: ...
187: }

Figure 13: The Scope Of Advice: Line 183 – 185 in glassbox.monitor.resource.JdbMonitor.

AspectJ Benchmark Suite and Design Patterns. In the Tetris, Dcm, and LawOfDemeter programs in the benchmark suite, XFindBugs detects 34 The Multiple Advice Invocation defects in total. For example, in Tetris, two data dependent advice Levels.before(): newgame() and Counter.before(): newgame() are declared.
6. RELATED WORK

Bugs are valuable for programmers to improve the software quality. XFindBugs detects such symptoms as a bad practice with priority Medium.

In NullCheck, XFindBugs finds six defects belonging to the The Infinite Loop bug pattern. We show the sample code of one defect in Figure 14. The method call of thisJoinPoint.getSignature() in line 78 is intercepted by the pointcut methodsThatReturnObjects in line 71. Therefore, an infinite calling loop (thisJoinPoint.getSignature() \rightarrow around(): methodsThatReturnObjects \rightarrow thisJoinPoint.getSignature()) is formed, which will result in a StackOverflowError. In this case, XFindBugs generates a warning and suggests programmers to add \texttt{within} (aspectname) to the pointcut designator, to prevent the defect.

![Figure 14: The Infinite Loop in NullCheck](image)

There are also several defects found by XFindBugs in the AspectJ benchmark suite and AspectJ implementation of design patterns. These defects are summarized in Table 3.

<table>
<thead>
<tr>
<th>Name</th>
<th>BAJMAT</th>
<th>BAJTSLA</th>
<th>BAJMOG</th>
<th>BAJAFBI</th>
<th>BAJOPR</th>
<th>BAJHVL</th>
<th>#All</th>
<th>%FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>AJHotdraw</td>
<td>8 (8)</td>
<td>0 (0)</td>
<td>2 (2)</td>
<td>0 (0)</td>
<td>0 (0)</td>
<td>10 (10)</td>
<td>37.5%</td>
<td></td>
</tr>
<tr>
<td>AJHSQILDB</td>
<td>142 (142)</td>
<td>0 (0)</td>
<td>3 (3)</td>
<td>1 (1)</td>
<td>1 (1)</td>
<td>0 (0)</td>
<td>147 (147)</td>
<td>0%</td>
</tr>
<tr>
<td>GlassBox</td>
<td>0 (1)</td>
<td>1 (1)</td>
<td>0 (0)</td>
<td>0 (0)</td>
<td>0 (0)</td>
<td>1 (1)</td>
<td>85.7%</td>
<td></td>
</tr>
<tr>
<td>AspectBenchmarks</td>
<td>34 (34)</td>
<td>0 (2)</td>
<td>0 (0)</td>
<td>0 (0)</td>
<td>0 (0)</td>
<td>34 (34)</td>
<td>5.5%</td>
<td></td>
</tr>
<tr>
<td>abcBenchmarks</td>
<td>60 (60)</td>
<td>0 (1)</td>
<td>0 (0)</td>
<td>0 (0)</td>
<td>0 (0)</td>
<td>3 (3)</td>
<td>1.56%</td>
<td></td>
</tr>
<tr>
<td>DesignPatterns</td>
<td>0 (0)</td>
<td>2 (10)</td>
<td>0 (0)</td>
<td>0 (0)</td>
<td>0 (0)</td>
<td>10 (10)</td>
<td>80%</td>
<td></td>
</tr>
<tr>
<td>RAI</td>
<td>244 (245)</td>
<td>3 (10)</td>
<td>5 (5)</td>
<td>1 (1)</td>
<td>1 (1)</td>
<td>33 (33)</td>
<td>8.2%</td>
<td></td>
</tr>
<tr>
<td>%FP</td>
<td>9.9%</td>
<td>88%</td>
<td>9%</td>
<td>0%</td>
<td>0%</td>
<td>8.2%</td>
<td>N/A</td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Defects found by XFindBugs in subject programs

5.6 False Positives

We manually check each defect reported by XFindBugs in the source code, and count the possible false positives. As shown in Table 3 (column %FP), the overall false positive ratio is 8.2%. For one specific bug pattern, the false positive ratio ranges from 0% to 88%. We can observe that most defects reported in AJHotdraw, AJHSQILDB and AspectJ benchmark suite are valid. While for the GlassBox application, the false positive ratio is very high (88%). This false positive ratio might be related to the complexity and variance of assignment instructions for different types inside the advice body.

7. CONCLUDING REMARKS

In this paper, we presented XFindBugs, an eXtended FindBugs for AspectJ, to detect potential defects in AspectJ software. In our current implementation, XFindBugs supports a catalog of 17 bug patterns, which cover common error-prone features of AspectJ programs. Our experience of using XFindBugs on several large-scale AspectJ applications highlights the practical issues of this tool. The experience also evidences the bug patterns presented in this do exist in real-world softwares.

In the future, we plan to identify more bug patterns in aspect-oriented software systems, and incorporate new bug detectors into XFindBugs. We would also like to investigate other sophisticated analyses (such as the verification approach [22, 28] and dynamic slicing [36]), to refine our existing detectors in XFindBugs to make them more accurate.

To encourage evaluation and further research in these and other directions, the source code of our bug detectors is available at http://cse.sjtu.edu.cn/~zhang/XFindBugs/

Acknowledgements. This work was supported in part by National High Technology Development Program of China (Grant No. 2006AA01Z158), National Natural Science Foundation of China (NSFC) (Grant No. 60673120, and Shanghai Pujiang Program (Grant No. 07pj14058). We would like to thank Qingzhou Luo, and Xin Huang for their valuable discussions on this work.
8. REFERENCES