Detection of Infinite Recursion in AspectJ Programs at Compile Time

Saeed Parsa$^1$ and Meisam Fathi Salmi$^2$

$^{1,2}$Iran University of Science and Technology
$^1$parsa@iust.ac.ir $^2$meisam.fathi@comp.iust.ac.ir

Abstract: This paper aims at automatic detection of infinite recursion at compile time in aspect-oriented programs. Infinite recursion is a known problem with aspect-oriented programming. If programmers do not take particular precautions, aspects that advise other aspects can easily and unintentionally lead to infinite recursion. The solution that is proposed in this paper informs programmers of aspects that lead to infinite recursion by showing compile time error messages and warnings. This paper, additionally, measures effectiveness of its proposed solution by applying it to several case studies. The solution that is proposed in this paper suggests that programming in aspect-oriented languages can be done more safely without restricting languages features or imposing runtime overhead.

Keywords: Infinite Recursion, Interprocedural Analysis, Aspect-Oriented Programming, Debugging Aspect-Oriented Programs

1 Introduction

A major problem with current mainstream aspect-oriented languages, such as AspectJ is that an aspect-oriented program can easily contain code that leads to unintentional infinite recursion. This unexpected infinite recursion is caused due to advice advising other advice or advising methods that they call. Recursion might be used intentionally in aspect-oriented programs by programmers for solving recursive problems, but in practice, it is mostly unintended, and consequently a programming error [8].

Programmers, particularly AspectJ programmers, use a particular workaround to avoid infinite recursion, such that for each aspect A, its pointcuts are conjuncted with a $\text{ctflow(within(A))}$ pointcut [8]. This workaround prevents infinite recursion caused by aspects advising themselves, but it has some drawbacks, too. Firstly, since $\text{ctflow}$ pointcuts cannot be matched statically, it brings on considerable runtime overhead [2, 3]. Secondly, even if the runtime overhead is not an issue, this approach requires programmers to follow conventions and conjunct each pointcut with $\text{ctflow(within(A))}$.

Testing is a common solution for revealing faults in all programming paradigms, including infinite recursion in aspect-oriented programs [9, 13]. However, most often, testing requires to be followed with further analysis to reveal the causes of the failures.

Bodden et al. [4] have proposed an extension to the AspectJ language, AspectJ*. This extension totally averts infinite recursion in aspects by layering aspects and preventing aspects from advising aspects in higher levels or aspects
in the same level. This approach prevents the infinite recursion problem, but it requires programmers to program in a new language, AspectJ\textsuperscript{*}. In order to help programmers, Bodden et al. have offered a refactoring for transforming AspectJ programs to AspectJ\textsuperscript{*} programs. Yet, it can be argued that refactoring existing AspectJ programs to AspectJ\textsuperscript{*} programs could be burdensome for large programs.

A totally different approach for preventing infinite recursion in AspectJ programs is proposed in [12] and has been implemented as a part of XFindBugs. XFindBugs examines the bytecode generated by a specific AspectJ compiler for detecting several special patterns that lead to faults in AspectJ programs. Some of the patterns that are examined by XFindBugs are patterns that lead to infinite recursion. XFindBugs detects some patterns that lead to infinite recursion, but a major difficulty with XFindBugs is its dependency on the bytecode generated by a specific AspectJ compiler, aje\textsuperscript{1}. In other words, XFindBugs is incapable of finding bugs in the compiled bytecode of AspectJ programs that are not compiled with the aje compiler.

In this paper a new approach for automatic detection of infinite recursion in aspect-oriented programs is presented. The proposed approach is based upon the static analysis of source code of aspect-oriented programs. The proposed approach does not require the programmer to insert any specific instruction within the program and does not impose any runtime overhead on the program execution.

The remaining parts of this paper are organized as follows: Section 3 delves into the proposed solution. In section 4 preliminary results of applying the proposed solution to several benchmarks are presented. Finally, section 5 deals with future work and conclusion.

2 Why infinite recursion occurs

There could be several join points within an advice. Also, there could be a number of method invocations within an advice. Each method body, accessed within an aspect contains its own join points. The join points inside the body of a method may include join points advised by another advice. Also an advice may advise another advice. Therefore, there could be a chain of advice applications and method calls. A method or an advice may appear more than once in the chain. If one of the method bodies or advice bodies calls a method that is already in the chain, or is advised by an advice that is already in the chain, a loop is formed.

As shown in Figure 1, a join point which is located in Method 1 is advised by Advice 1. Inside the body of Advice 1, Method 2 is called. Another join point which is located in Method 1 is advised by a second advice, Advice 2. These two methods and two advice bodies form a chain.

\footnote{http://eclipse.org/aspectj}
If a cycle in the chain is formed only by the methods, the problem is to detect recursive method calls, which is already solved [6]. But, if the cycle contains at least one advice body, a recursion detection technique different from the ones applied for detecting recursive calls is required.

There are two main reasons behind the existence of cycles in the chain of method calls and aspect applications in aspect-oriented programs, as follows:

1. Aspect-oriented languages are designed and implemented for dealing with crosscutting concerns. Such concerns show up in many different places in the code. In order to implement crosscutting concerns advices that implement them should advise numerous different join points in the code. It is very difficult for programmers to keep track of these join points.

2. Current mainstream aspect-oriented languages use structures for defining pointcuts that are fragile [7], that is, with a local change in the source code of an aspect-oriented program, semantics of non-local aspects or classes may change.

3. The increasing number of pointcuts, capturing crosscutting concerns of aspect oriented languages, makes it difficult for the programmer to keep track of the join points and detect advices advising each other in a cycle.

4. Careless modification of an aspect oriented program semantics may result in new joinpoints, adding advices and method calls which cause a cycle in the chain of the program advice applications and method calls.

3 The proposed approach

The approach that is proposed in this paper consists of two parts. The first part, which is described in Section 3.1, is concerned with transforming an aspect-oriented program to a model that can be used for analyzing the source code of aspect-oriented programs. The second part, which is described in Section 3.2, is concerned with detection of infinite recursion in the transformed model.

3.1 The program model

The proposed approach of this paper is based on two assumptions:

1. Advice bodies and method bodies are equivalent, except for the fact that each advice body has a pointcut associated with it.

2. All join points reside in methods.

The outcome of the first assumption is that when a join point is captured by a pointcut and its relevant advice is executed, it can be assumed that a method call has happened, and the method that is executed is the body of the advice.

The second assumption needs more elaboration. A join point can reside in any of the following places: inside a method, inside an advice, inside an initialization block, inside a class constructor, inside a static initialization block,
Figure 2: A Sample Snippet of an Aspect-Oriented Program

at a static field initialization, and at a non-static field initialization. When a join point does not reside in a method, several clarifications are required to keep the assumption true. These clarifications are as follows:

1. **Clarifying join points inside an advice:** The first assumption about the equivalence of methods and advice makes this clarification straightforward. Joint points that are inside an advice whose signature is \( \text{advice} : \text{param}_1 \times \text{param}_2 \times \cdots \times \text{param}_n \rightarrow \text{out} \) are supposed to reside in a method with the same name whose signature is \( \text{advice} : \text{param}_0 \times \text{param}_1 \times \text{param}_2 \times \cdots \times \text{param}_n \rightarrow \text{out} \). This method has an extra \( \text{param}_0 \) which contains the context of program at the captured join point.

2. **Clarifying join points inside an initialization block:** Joint points that are inside an initialization block are supposed to reside in a special \( \langle \text{init} \rangle \) method which has no parameters.

3. **Clarifying join points inside a constructor:** Join points that are in a constructor whose signature is \( \text{new} : \text{param}_1 \times \text{param}_2 \times \cdots \times \text{param}_n \rightarrow \text{out} \) are assumed to reside in a special method whose signature is \( \langle \text{init} \rangle : \text{param}_1 \times \text{param}_2 \times \cdots \times \text{param}_n \rightarrow \text{out} \).

This assumption implies that for classes with more than one constructor, there are several \( \langle \text{init} \rangle \) methods with different signatures in the program model. The parameter lists of each \( \langle \text{init} \rangle \) method is defined by its corresponding constructor. All constructors are supposed to call the special method \( \langle \text{init} \rangle \) at their entry point before other statements inside their bodies. If there is a default constructor (i.e. constructor with no parameters), it is supposed that its body is merged with the body of \( \langle \text{init} \rangle \).

<table>
<thead>
<tr>
<th>Line#</th>
<th>Join Point</th>
<th>Residue</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>set(theNumber)</td>
<td>&lt;static&gt;()</td>
</tr>
<tr>
<td>2</td>
<td>new(Random)</td>
<td>&lt;init&gt;(int)</td>
</tr>
<tr>
<td>4-8</td>
<td>execution(maxInArray)</td>
<td>maxInArray(int)</td>
</tr>
<tr>
<td>5</td>
<td>get(array.length)</td>
<td>maxInArray(int)</td>
</tr>
<tr>
<td>6</td>
<td>call(max(double, double))</td>
<td>maxInArray(int)</td>
</tr>
</tbody>
</table>
4. Clarifying join points inside a static initialization block: Joint points that are located inside a static initialization block are supposed to reside in a special static method, \( (\text{static}) \), which takes no input parameters. If there are more than one static initialization block in a class, their corresponding join points are supposed to reside in \( (\text{static}) \) method in the order that they appear in the source code.

5. Clarifying join points at a static field initialization: Joint points that are associated with initialization of static fields are supposed to reside in the special method \( (\text{static}) \). The order of their appearance in \( (\text{static}) \) is the same as the order in which they appear in the source code.

6. Clarifying join points at a non-static field initialization: Joint points that are associated with initialization of non-static fields are supposed to reside in the special method \( (\text{init}) \). The order of their appearance in \( (\text{init}) \) method is the same as the order in which they appear in the source code.

Figure 2 is a snippet from the Null-Check benchmark that is distributed with AspectBench Compiler. Clarification for static methods and static method initializers for this snippet are shown in Table 1.

These assumptions are used to extract a program model \( P \) from the source code. This model contains, a set \( B_P \) of all methods and advice bodies defined in \( P \), a set \( JP_P \) of join points defined in \( P \), a relation \( C_P \subseteq B_P \times B_P \) of all method calls in \( P \), a relation \( A_P \subseteq JP_P \times B_P \) which indicates join points advised by each advice, and a relation \( R_P \subseteq JP_P \times B_P \) which indicates residues for each join point. Since this model assumes that methods and advice are equivalent, not only does the relation \( C_P \) contain method calls, but also it contains all advice bodies that are woven into join points. The \( B_P \) itself is the union of \( MB_P \) and \( ABP_P \), respectively the set of all method bodies, and the set of all advice bodies defined in \( P \).

\[
\]

This model is used for constructing an extended call graph, \( ECG_P = (N_P, E_P) \), in which method and advice bodies form vertices of the graphs. An edge \( e = (b_1, b_2) \) belongs to this graph provided that one of the following conditions is satisfied:

- There are method bodies \( b_1 \) and \( b_2 \) such that \( b_1 \) calls \( b_2 \).
- There is an advice body \( b_1 \) and a method body \( b_2 \) such that \( b_1 \) invokes \( b_2 \).
- There is a method body \( b_1 \), an advice body \( b_2 \), and a join point \( jp \), such that \( jp \) resides in \( b_1 \) and is advise by \( b_2 \).
- There are advice bodies \( b_1 \) and \( b_2 \), and a join point \( jp \) such that \( jp \) resides in \( b_1 \) and is advised by \( b_2 \).
\[ (b_1, b_2) \in ECG_F \iff (\exists b_1 \in B_F \land \exists b_2 \in B_F : (b_1, b_2) \in C_F) \]
\[ \lor (\exists jp \in JF_P : (jp, b_1) \in R_F \land (jp, b_2) \in A_F) \] (2)

Consequently, there is an edge \((b_1, b_2)\) in \(ECG\) if and only if \(b_2\) is reachable from \(b_1\) via a method call or an advice application execution. For finding all edges it is necessary that all join points and their residues be identified.

For extracting this model AspectBench Compiler abc [1], which is an extensible AspectJ compiler, is used. This compiler is capable of running user defined passes and analyses, i.e. it is possible to define your desired passes and analysis and add them to abc.

3.2 Cycle detection in the extended call graph

Each edge in the extended call graph is labeled with one of the following labels:

c : An edge labeled with c is guaranteed to be executed each time its containing method is executed.

u : An edge labeled with u maybe executed each time its containing method is executed, this, however, is not guaranteed.

Let \(CFG_b = (N_b, E_b, n_{entry}, n_{exit})\) be the control flow graph of method \(b \in B_F\), in which \(N_b\) is the set of nodes, \(E_b\) is the set of edges, \(n_{entry}\) is the entry node, and \(n_{exit}\) is the exit node. Let’s suppose that edge \(e = (b, b')\) is due to node \(n\) in \(CFG_b\), i.e. statement \(n \in N_b\) is a method call statement that invokes \(b'\). Additionally, let \(\triangleright\) indicate the dominator relation, such that \(n \triangleright n'\) indicates \(n\) dominates \(n'\).

An edge \(e = (b, b') \in ECG_F\) is labeled \(c\) if and only if \(b \triangleright b_{exit}\), else it is labeled \(u\).

In order to detect cycles in the extended call graph, the default cycle detection algorithm provided by abc is used. The cycle detection algorithm is applied to the call graph twice. The first application of the algorithm detects cycles whose edges are labeled with \(c\). This application of the algorithm detects cycles which are guaranteed to lead to infinite recursion. The second application of the algorithm detects all cycles regardless of the labels of their forming edges. This second application of the algorithm detects cycles which may or may not lead to infinite recursion. In theory, not only does this application detect cycles that are detected by the previous application, and consequently are programming errors, but also detects cycles that might be intentionally developed by programmer. In practice, however, the likelihood of intentional creation of advice that participates in a cycle is very low.

4 Preliminary results and discussion

The solution that is proposed in this paper has been applied to a variety of benchmarks collected from different sources. The Logging benchmark is an implementation of the logging aspect that is introduced by Laddad in [8]. The Profiling benchmark is one of the several profiling aspects that are introduced
in [10]. The Null-Check and Law of Demeter benchmarks are benchmarks that are distributed with abc [1] and are used by Dafur et al. [5]. The Non-Negative and Instrumentation benchmarks are used by Richard et al. [11]. These benchmarks have been applied to some base applications, among which the Telecom benchmark is distributed with AspectJ, and the Stack and Account applications are benchmarks that are used by Richard et al. [11]. Comparing to other benchmarks, the Profiling and Logging benchmarks have two distinct characteristics. Firstly, they have pointcuts that have the possibility to capture join points scattered in different packages and classes. As shown in line 5 of Figure 3, the Profiling benchmark defines a pointcut that captures all methods calls to the `toString()` methods.

Secondly, these benchmarks invoke several methods, which might be advised by other aspects, in the body of their advice. As shown in lines 8-11 of Figure 3 the Profiling benchmark invoke several methods over the `totalCount` object and other objects in one of its advice bodies. Containing pointcuts that capture join points from different classes and invoking several methods in their body increases the likelihood of forming infinite recursion.

```
@Aspect
public class ToStringCountingAspect {
  private Map totalCounts = new HashMap();
  private int myCount = 0;

  @Pointcut(myCall() : call(String MyClass.toString()))
  public void allCalls() { allCalls(); }

  @Before(): myCall() { myCount++;

  @After(): allCalls() {
    Class c = thisJPSF.getSignature().getDeclaringType();
    Integer i = (Integer) totalCounts.get(c);
    if (i != null) totalCounts.put(c, new Integer(i.intValue()+1));
    else totalCounts.put(c, new Integer(1));
  }
}
```

Figure 3: The Source Code of Profiling Benchmark

Table 2: Number of Faults that has been Detected in Different Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Bank</th>
<th>Account</th>
<th>Telecom</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logging</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Profiling</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Null-Check</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Law of Demeter</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Inversion</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Design by Contract</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

Table 2 represents the result of applying the proposed approach to the mentioned benchmarks. The first row in this table shows the name of the base application that is used. Several benchmarks have been applied to each base application. These benchmarks are marked in rows 2 to 9 of Table 2. The last four rows show the results of applying the proposed approach to the benchmarks.
The Number of Edges in ECG row shows the number of edges that have been constructed in the extended call graph during analysis of benchmarks.

In Table 2, the Certain Cycles and Non-Certain Cycles rows show the number of cycles that have been identified as cycles that certainly happen, and the number of cycles that might happen, respectively. As mentioned in Section 3.2 certain cycles contain edges labeled with c while non-certain cycles contain edges labeled with both c and u.

All detected cycles have been examined manually to see if they really lead to infinite recursion. The last row of Table 2 shows the correctness of the solution in terms of the the number of False Negatives. As represented in Table 2, while none of the benchmarks include a fault individually, infinite recursion might occur when more than one benchmark is applied to a base application. As it was expected, when benchmarks contain pointcuts that capture join points located in different classes, in this case for Logging and Profiling benchmarks, advice advising other advice may lead to infinite recursion. Putting Logging and Profiling benchmarks away, no combination of other benchmarks has ever resulted in infinite recursion, mainly because other benchmarks have pointcuts that strictly restrict join points that they capture by explicitly identifying packages and classes that should be advised.

Additionally, the proposed approach is not capable of detecting infinite recursion when the Law-of-Demeter benchmark is woven to the base application along with Logging and Profiling benchmarks. The authors believe that this inability to detect infinite recursion when the Law-of-Demeter benchmark is applied to the base application is due to presence of dynamic pointcuts in this benchmark which are evaluated at runtime.

5 Conclusion and future work

The overall experiment suggests that despite the considerable number of False-Positive results, the solution can lead to development of safer programs, however, it suffers from two main restrictions that we would like to redress in our future work. First of all, the cycle detection algorithm is not efficient enough to be used for large programs. In a real world program, the number of vertices in the extended call graph of the program can be drastically high. This makes the analysis time of the programs unreasonably long.

Then after, as described in Section 4, the proposed solution cannot always detect infinite recursion correctly. Lower False-Negative rates are achievable via more sophisticated analysis of the source code. This apparently achievable via predicting join points that are captured by each dynamic pointcut.

Acknowledgment

The authors would like to thank all the students at the Parallel and Concurrent Processing Research Laboratory, Eric Bodden, who was an ocean of knowledge whenever it came to abe.
References


