Detecting Precedence-Related Advice Interference

Maximilian Störzer and Robin Sterr
University of Passau
{stoerzer, sterr}@fmi.uni-passau.de

Florian Forster
University of Hagen
Florian.Forster@fernuni-hagen.de

Abstract

Aspect-Oriented Programming (AOP) has been proposed in literature to overcome modularization shortcomings such as the tyranny of the dominant decomposition. *However, the new language constructs introduced by AOP also raise new issues on their own–one of them is potential interference among aspects. In this paper we present an interference criterion to detect and thus help programmers to avoid advice order related problems.*

1 Motivation

Aspect-Oriented Programming (AOP) as introduced in [5] has been promoted in recent years as a mechanism to overcome the *tyranny of the dominant decomposition*. Even in well-designed systems some requirements cannot be localized as a separate module and are thus called *crosscutting concerns*. This code typically is hard to maintain as changes to it result in invasive changes of the system.

Today aspect-oriented extensions are available for most main stream languages, although the Java extension AspectJ[1] [4] still is the most popular AO language. AO languages as understood in this paper share a common core principle: a new kind of module called *aspect* contains definition of behavior (called *advice*) and a specification where this behavior should be executed (called *pointcuts*). Pointcuts are *quantified statements* over a program selecting a set of well-defined points during the execution of a program (called *joinpoints*). Examples for joinpoints are method calls or field access. AspectJ follows this principle and we will use it as example in the following.

AOP is not without problems. Recently there was an interesting discussion on the AspectJ mailing list illustrating a major AOP problem: *aspect interference*. An AspectJ user had the following problem migrating his system to a new version of the AspectJ compiler[2]: *"What I am seeing ... is that my aspects that previously worked ... are no longer working. ... I can not stress enough that the only change was my migration from 1.2 variants of AspectJ and AJDT to the newest versions when this started to occur."* What changed in between these two compiler versions?

In the absence of explicit user-defined aspect ordering advice precedence for two pieces of advice can be *undefined*. The above problem could be tracked down to a change in *compiler specific precedence rules*. The new compiler chose a different order in cases where advice order was undefined, finally resulting in a program failure as advice is not commutative in general. The advice-precedence related problem reported on the mailing list is a special case of a problem known as the *aspect interference problem* in AOP research [1]. We will term this above mentioned special case *the advice precedence problem* in this paper. While one could argue that programmers should not rely on a particular order picked by a compiler, the problem is more substantial as it is only triggered by *combining potentially independently developed modules* or even *evolution of some modules*. For example assume a new method is added to the system where now two pieces of advice from two formerly non-conflicting aspects apply. Each system modification thus always requires to *recheck* whether applied aspects interfere with each other in the new version. Manual checking is tedious and error prone, as conflicts are not obvious.

Manually maintaining aspect precedence among a large number of aspects is hard, as a multitude of potential conflicts has to be examined by the programmer. We argue that tool support can leverage this problem by automatically detecting if advice precedence has to be defined. In this paper we therefore define an interference criterion, which we also implemented prototypically.

2 Example

We will use the simple `Telecom` application which is part of the AspectJ distribution to illustrate the advice precedence problem in this paper. The discussion of this example will also give a short introduction to AspectJ for readers unfamiliar with the language. The `Telecom` appli-

[1] We use the AspectJ language version 1.2 for this paper.

[2] [aspectj-users] AJDT 1.3 and aspectj; thread started by Ronald R. DiFrango on Oct. 10th, 2005

21st IEEE International Conference on Automated Software Engineering (ASE'06)
0-7695-2579-2/06 $20.00 © 2006 IEEE

Listing 1. `Timing` **and** `Billing` **aspects and** `Timer`**-class of** `Telecom` **example.**

```
1  public aspect Timing {
2    private Timer Connection.timer=new Timer();
3    public Timer getTimer}(Connection conn)
4    { return conn.timer; }
5    pointcut endTiming(Connection c):
6      call(void Connection.drop()) && target(c);
7    after(Connection c): endTiming(c)
8    { getTimer(c).stop();
9      c.getCaller().totalConnectTime
10       += getTimer(c).getTime();
11     c.getReceiver().totalConnectTime
12       += getTimer(c).getTime(); }
13   ...
14 }
15 public aspect Billing {
16   /* declare precedence: Billing, Timing; */
17   after(Connection conn):
18     Timing.endTiming(conn)
19   { long time = Timing.aspectOf()
20       .getTimer(conn).getTime(); ... }
21   ...
22 }
23 public class Timer {
24   public long startTime, stopTime;
25   public void start}()
26   { startTime = System.currentTimeMillis();
27     stopTime = startTime; }
28   public void stop}()
29   { stopTime = System.currentTimeMillis(); }
30   public long getTime}()
31   { return stopTime - startTime; }
32 }
```

cation models a telecommunication administration system. The base application is extended with two aspects called `Timing` and `Billing`. The `Timing` aspect keeps track of the duration of a phone call, while the `Billing` aspect uses this information to calculate the amount of money that customers are charged. Originally, aspect `Billing` contains an AspectJ statement to explicitly define that `Billing` has higher precedence than `Timing` (`declare precedence`, line 16). However, we removed this statement to demonstrate interference problems.

Figure 1 shows the relevant code[3] of the `Telecom` example. Consider pointcut `endTiming` (line 5), which uses the `call` keyword of AspectJ to select joinpoints representing calls to the `Connection.drop()` method. In the following c is the `Connection`-object `drop()` is called on. The `target` keyword is used to expose c to advice bound to this pointcut.

Two pieces of `after`-advice reference pointcut `endTiming` (lines 7 and 17) and are executed *immediately after* the call to `drop()` returns. As c is exposed by the pointcut, both pieces of advice can access c to perform their calculations. Besides `after`-advice, AspectJ also defines two other *kinds* of advice: `before`- and `around`-advice[4], allowing to add behavior *before* or *instead of* a selected joinpoint, respectively.

The `Telecom` example also uses *inter type declarations* to add new members to existing classes. The declaration `Timer Connection.timer = new Timer()` (line 2) for example adds and initializes a field `timer` in the `Connection` class. Compared to traditional member declarations inter type declarations use qualified names to specify the target class the new member should be a part of.

Ending a phone call is modeled by calling `hangUp()` on a `Customer`-object which finally results in a call to `drop()` on the `Connection` object. The pointcut `endTiming` binds the `after`-advice of both the `Timing` and `Billing` aspects to the joinpoint representing the `drop()` call. As can be seen, `Timing` saves the end time of the phone call (`getTimer(c).stop()`, line 8) and `Billing` uses this information to calculate the amount of money the caller is charged (`getTimer(conn).getTime()`, line 20).

The `declare precedence` statement in the original example guarantees that the advice defined in the `Timing` aspect is executed before the advice defined in the `Billing` aspect. However, if this statement is missing, the compiler is free to choose the opposite order as in this case advice precedence according to the language specification is undefined. In this case the `Billing`-advice will always receive 0 when calling `getTime()` on the shared `Timer`-object, as the observant reader may verify. System functionality is broken; here the order in which both actions are performed is obviously relevant.

For this example, dependence between the two aspects is easy to see and thus to fix. In large, team-developed projects non-trivially interfering aspects might be developed by different programmers, making it hard to even notice interference. The resulting errors are cumbersome to detect, as the interference may be well-hidden.

3 Advice Interference

In the `Telecom` example introduced in section 2 `Billing` uses information written by `Timing`. A "read from relation" is also well-known from transaction serialization theory for databases. There, two transactions T_1 and T_2 *conflict* if they both access a common data object and at least one of them modifies this object. If two transactions conflict, they have to be serialized to maintain database consistency. Similarly, data flow between two pieces of advice

[3] Due to space limitations, we only show relevant code here.

[4] For `around`-advice, the original behavior at the joinpoint can be called by using the `proceed` keyword.

is relevant, although data flow analysis alone is not sufficient. Additionally the control flow can also be altered by advice, and thus has to be analyzed as well.

To formulate the data flow interference criterion, we need to know which parts of the system state are used by a piece of advice. This is captured by the def- and use-sets for a piece of advice.

Definition 3.1 ($def()$ and $use()$-sets) *Let 'm' be a method or a piece of advice. Let '$decl$' be the unique source location of a variable declaration. Let N_t be the set of call targets for all call sites and A_t the set of all pieces of advice attached to a joinpoint in m's lexical scope. Then $def()$ and $use()$ are defined as follows:*

$$\begin{aligned} def(m) = &\{(a, decl(a)) \mid a \textit{ appears as l-value in } m\} \\ &\cup \{(a.x, decl(a)) \mid a.x \textit{ appears as l-value in } m\} \\ &\cup \bigcup_{n_t \in N_t} def(n_t) \cup \bigcup_{a_t \in A_t} def(a_t) \\ use(m) = &\{(a, decl(a)) \mid a \textit{ appears as r-value in } m\} \\ &\cup \{(a.x, decl(a)) \mid a.x \textit{ appears as r-value in } m\} \\ &\cup \bigcup_{n_t \in N_t} use(n_t) \cup \bigcup_{a_t \in A_t} use(a_t) \end{aligned}$$

Traditionally the def- and use-sets of a method (and similarly advice) are defined as the set of memory locations defined (or written) and used (or read) by a method, respectively. We define $def()$ and $use()$-sets semi-formally based on the statements contained in a method or advice.

Definition 3.1 states that all identifiers (both qualified and unqualified) used as l-values are added to the $def()$-set, while all identifiers used in expressions or method calls as parameters are added to the $use()$-set (as they are implicitly used as r-values). Note that we add the $def()$- and $use()$-sets of called methods and attached advice as well, as we have to analyze data access in the complete control flow of potentially conflicting advice, including subsequently called methods. The recursion in definition 3.1 terminates for methods without call site and attached advice. Recursive methods are no problem here, as a single analysis of a given method is sufficient to collect all names appearing either as l- or r-value.

Analyzing all pieces of advice is not necessary, as only a small set of available pieces of advice is relevant. We define advice data dependence for relevant advice in the following.

Definition 3.2 (Relevant Advice) *Two pieces of advice a_1 and a_2 are* relevant*, if they are defined in different aspects, apply at at least one common joinpoint and are either of the same kind or at least one of them is* `around`*-advice.*

Definition 3.3 (Advice Data Dependence) *Let a_1 and a_2 be two relevant pieces of advice. Let '$objects$' denote the objects a reference may refer to, '$formals(a)$' be the formal parameters of a piece of advice 'a', and '$actuals($*`proceed`*$)$' the actual parameters of the call to* `proceed`*. Then a_1 and a_2 are* data dependent *on each other if for $i, j \in \{1, 2\}, i \neq j$ either*

$$\begin{aligned} &(a.x, decl(a)) \in def(a_i) \wedge (b.x, decl(b)) \in (def(a_j) \cup use(a_j))) \\ &\quad \Rightarrow objects(a, decl(a)) \cap objects(b, decl(b)) \neq \emptyset \end{aligned}$$

or, if a_i is `around`*-advice,*

$$\begin{aligned} &formals(a_i) \not\equiv actuals(\texttt{proceed}) \vee \\ &formals(a_i) \cap def(a_i) \neq \emptyset \vee \\ &a_i \textit{ returns a different value than } \texttt{proceed}(\ldots). \end{aligned}$$

The first property checks if one advice reads data from the other, similar to the criterion stated for transactions. The second criterion however handles the special case of `around`-advice which can easily modify or redefine the actual parameters or return values of the `proceed`-call. The second property explicitly captures these cases.

Advice can modify control flow such that execution of pieces of advice with lower precedence applying at the same joinpoint is prevented (e.g. by throwing an exception). In this case program semantics again depend on advice precedence; order thus has to be explicitly stated. We define control dependence and finally advice interference as follows.

Definition 3.4 (Advice Control Dependence) *Let a_1 and a_2 be two relevant pieces of advice. a_i is* control dependent *on a_j (for $i, j \in \{1, 2\}, i \neq j$), if a_j explicitly throws an exception which is not handled in the advice body of a_j or if a_j is* `around`*-advice and* `proceed` *is not called exactly once in it's control flow or any exception thrown by* `proceed` *is handled by the advice.*

Note that demanding that `proceed` is called *at least once* is not sufficient as multiple calls to `proceed` subsequently result in multiple executions of respective methods and advice so likely changing system semantics.

Definition 3.5 (Advice Interference) *Two relevant pieces of advice a_1 and a_2 interfere, if a_1 is* data or control dependent *on a_2 or vice versa.*

Finally we define aspect interference based on conflicting advice.

Definition 3.6 (Aspect Conflict) *Let A_1 and A_2 be two aspects. Then A_1 and A_2* conflict*, if two pieces of advice $a_1 \in A_1, a_2 \in A_2$ exist such that a_1 and a_2 interfere and precedence of A_1 and A_2 is undefined.*

Example 3.1 (`Telecom`) *We apply definition 3.6 to the* `Telecom` *example from section 2. Let a_1 be the* `after`*-advice in* `Timing`*, a_2 the* `after`*-advice in* `Billing`*. As the reader may verify, a_1 and a_2 are relevant and access the same* `Timer`*-object o_{tim} associated*

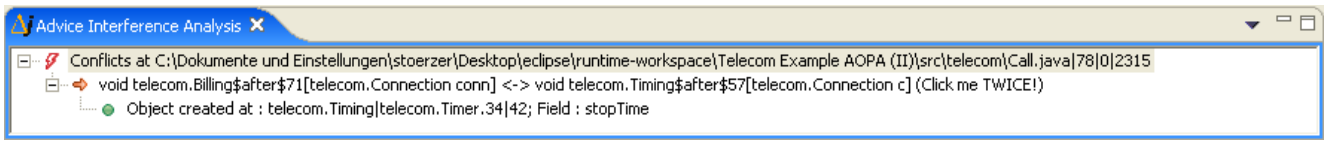

Figure 1. Report for the `Telecom` example.

with the current connection through their respective call to `getTimer(Connection)`. a_1 calls o_{tim}.`stop()`, thus setting o_{tim}.`stopTime`, i.e. o_{tim}.`stopTime` $\in def(a_1)$. a_2 in turn calls o_{tim}.`getTime`, so reading o_{tim}.`stopTime`, i.e.o_{tim}.`stopTime` $\in use(a_2)$. So there is a data dependence between a_1 and a_2 on o_{tim}.`stopTime`. As a consequence a_1 and a_2 interfere and our criterion discovers that aspects `Timing` and `Billing` have to be explicitly ordered, as is the case in the original version of the `Telecom` example.

We prototypically implemented our criterion as an Eclipse plug-in, by analyzing control flow graphs to capture exceptional control flow and using pointer analysis to approximate function *objects*. Figure 1 shows a screenshot of the results presented by our tool when analyzing the `Telecom` example.

4 Related Work and Conclusions

Although the problem of aspect interference has been recognized by researchers, there are still only few approaches in literature addressing this problem. In [1] the aspect interaction problem is discussed in a position paper, although on a more abstract level and different context. While this work contains an interesting discussion of problem itself, a solution is only briefly outlined. In [6] a reflective aspect-oriented framework is proposed which allows users to visually specify aspect-base and aspect-aspect dependences using the Alpheus tool. While the framework offers a more abstract view on aspects and provides a richer set of conflict resolution rules than AspectJ, conflict detection is manual.

In [2, 3], a non-standard but base-language independent aspect-oriented framework, including support for conflict detection and resolution, is discussed. The presented conflict resolution mechanisms are more powerful than the `declare precedence` construct of AspectJ. However the presented model does not handle `around`-advice and bases conflict detection on multiple pieces of advice applying to a single joinpoint only. Our method in contrast explicitly analyzes advice for commutativity thus reducing the number of false positives.

In [7] Rinard et. al. propose a combined pointer and effect analysis to classify aspects by their effects on the base system. While we use a similar underlying analysis, the target of our work differs from their's.

To summarize, the contributions of this paper are threefold. First, we provided an in depth analysis of the advice precedence problem and demonstrated its relevance. Second, we defined an interference criterion to check for relevant undefined advice precedence. Third and finally we implemented this criterion prototypically, although a discussion of the details had to be omitted due to space restriction. To conclude our approach supports programmers working with AO systems who have to deal with the advice precedence problem by detecting unexpected side effects during system construction and evolution.

References

[1] L. M. Bergmans. Towards detection of semantic conflicts between crosscutting concerns. *Proceedings of workshop AAOS 2003: Analysis of Aspect-Oriented Software, held in conjunction with ECOOP 2003*, 2003.

[2] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *GPCE '02: The ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, pages 173–188, London, UK, 2002. Springer-Verlag.

[3] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 141–150, New York, NY, USA, 2004. ACM Press.

[4] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[5] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[6] J. L. Pryor and C. Marcos. Solving conflicts in aspect-oriented applications. In *Proceedings of 4th Argentine Symposium in Software Engineering*, Buenos Aires, Argentina, September 2003.

[7] M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 147–158, New York, NY, USA, 2004. ACM Press.