Detecting Precedence-Related Advice Interference

Technical Report TR #0607, University of Passau, Computer Science Department, Passau, Germany, July 2006

Maximilian Stoerzer 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 focus on a special case of this problem and demonstrate how undefined advice precedence can easily jeopardize correctness of a program. We present an interference analysis to detect and thus help programmers to avoid advice order related problems.

1 Motivation

Aspect-Oriented Programming (AOP) as introduced in [10] has been promoted in recent years as a mechanism to overcome the *tyranny of the dominant decomposition* [15]. Even in well-designed systems it can happen that some requirements cannot be implemented well-localized as a separate module. Such requirements are called *crosscutting concerns*, as their implementation *scatters* code cross several modules. This scattered code typically is hard to maintain as changes to it result in invasive changes to major parts of the system. Classical examples are tracing or authorization.

Today aspect-oriented extensions are available for most main stream languages, although the Java extension AspectJ[1] [9] 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 for transaction control and database connections 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. In the same thread, AspectJ developers state that no guarantee can be given on any order picked by the compiler for undefined precedence, even that the order can change arbitrarily among different compiler versions or even for different compiler runs.

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 [4]. 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.

First, in team developed projects different programmers potentially develop and test aspects *independently* of each other. As a consequence these programmers in general are not aware of other aspects let alone whether or not their aspects interfere. Second, due to the design of pointcuts as quantified expressions over a program, *evolution* of the base

[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

program may result in introduction of aspect interference some time *after* aspects have been applied and tested. 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 check whether applied aspects interfere with each other in the new version. Doing this manually is tedious and error prone, as conflicts are not obvious. And finally manually maintaining aspect precedence among a large number of aspects can be hard, as a multitude of potential conflicts has to be examined by the programmer. Most aspects however might not conflict at all. Thus the problem can easily be neglected as an irrelevant effort ("Aspects don't interfere."), leaving system semantics undefined if aspects conflict.

We argue that *tool support* is necessary to make programmers aware of interfering aspects at compile time. In this paper we analyze the problem of undefined aspect precedence in detail and define advice order related aspect interference. Further on we propose a method to automatically determine a set of potentially interfering aspects based on static analysis of a system. Although we use the AspectJ programming model–focusing on advice, pointcuts and (statically resolvable) joinpoint matching–our approach is applicable to languages based on a comparable programming model as well.

The contributions of this paper are threefold. First, we discuss the *problem of aspect interference* which up to now has been mostly neglected by aspect research. Second, we present an *interference criterion* allowing to detect situations where establishing a defined advice precedence is necessary. Third, we used program analysis techniques to *implement this criterion*.

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` application 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. In the original example, aspect `Billing` contains an AspectJ statement to explicitly define that `Billing` has higher precedence than `Timing`: `declare precedence: Billing, Timing;` However, we will remove this statement to demonstrate interference problems.

Figure 1 shows the `Timing` and `Billing` aspects[3] of the `Telecom` example. Consider the definition of the pointcut named `endTiming` in `Timing`, which uses the `call` keyword of AspectJ to select joinpoints representing calls to the `drop` method of the `Connection`-class. 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.

There are two pieces of `after`-advice defined in the `Timing` and `Billing` aspects referencing pointcut `endTiming` (shown in bold). These pieces of `after`-advice 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 `before`- and `around`-advice, allowing to add behavior *before* or *instead of* a selected joinpoint, respectively. We call this the advice *kind*. For `around`-advice, the original behavior at the joinpoint can be called by using the `proceed` keyword.

Besides pointcuts and advice the `Telecom` example also uses *inter type declarations*, which allow to add new members to existing classes. The declaration `Timer Connection.timer = new Timer()` for example adds and initializes a field `timer` in the `Connection` class. The difference compared to traditional member declarations is the qualified name, specifying the target class the new member should be a part of. This short introduction to AspectJ should suffice for this paper, for details we refer the interested reader to [9] and the AspectJ manuals.[4]

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 (shown in bold in figure 1) to the joinpoint representing the `drop()` call. As can be seen, `Timing` saves the end time of the phone call (`getTimer(c).stop()`) and `Billing` uses this information to calculate the amount of money the caller is charged (`getTimer(conn).getTime()`). The `Timer`-class is shown in figure 2.

```
public class Timer {
  public long startTime, stopTime;
  public void start()
  { startTime = System.currentTimeMillis();
    stopTime = startTime; }
  public void stop()
  { stopTime = System.currentTimeMillis(); }
  public long getTime()
  { return stopTime - startTime; }
}
```

Figure 2. `Timer` **class.**

The `declare precedence` statement in the original example guarantees that the advice defined in the `Tim-`

[3] Due to space limitations, we omit the base code here.

[4] http://www.eclipse.org/aspectj/

```
public aspect Timing {
  private Timer Connection.timer = new Timer();
  public Timer getTimer(Connection conn)
  { return conn.timer; }

  after (Connection c): call(
    void Connection.complete()) && target(c)
  { getTimer(c).start(); }

  pointcut endTiming(Connection c):
    call(void Connection.drop()) && target(c);

  after(Connection c): endTiming(c)
  { getTimer(c).stop();
    c.getCaller().totalConnectTime
      += getTimer(c).getTime();
    c.getReceiver().totalConnectTime
      += getTimer(c).getTime(); }
  ...
}
```

```
public aspect Billing {
  /* declare precedence: Billing, Timing; */
  private Customer Connection.payer;
  public Customer getPayer(Connection conn)
  { return conn.payer; }
  after(Customer cust) returning (Connection conn):
    args(cust, ..) && call(Connection+.new(..))
  { conn.payer = cust; }
  public abstract long Connection.callRate();
  after(Connection conn):
    Timing.endTiming(conn)
  { long time = Timing.aspectOf()
      .getTimer(conn).getTime();
    long rate = conn.callRate();
    long cost = rate * time;
    getPayer(conn).addCharge(cost); }
  public long Customer.totalCharge = 0;
  public void Customer.addCharge(long charge)
  { totalCharge += charge; }
  ...
}
```

Figure 1. `Timing` **and** `Billing` **aspects of Telecom example.**

ing 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. As a consequence 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 the necessary explicit ordering is easy to add. This is not the case in general. In large, team-developed projects aspects interfering non-trivially 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. The missing ordering might also fail to produce a failure if the compiler by chance chooses the "right" order. In this case even thorough testing fails to detect a potential problem. Note that this is not a weakness of testing, but rather a principal problem of changing semantics due to a different compiler run—in the first version there simply *is* no problem a test could reveal. This however might change with the next compiler version or even with the next compilation of the system, if the new compiler chooses the "wrong" order.

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. In this context, 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.

Analysis of data flow between two pieces of advice is the first cornerstone of our analysis. Besides data flow, multiple pieces of advice at a single joinpoint can also interfere, if they potentially prevent execution of subsequent advice. This is the case, for example, if advice throws an exception or around-advice does not call proceed. Analysis of control dependences thus is the second cornerstone of our analysis. We will declare advice interference based on these two properties. If two pieces of advice conflict, aspect precedence has to be explicitly declared to avoid undefined system semantics.

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:*

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

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

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. Note that we assume that nested statements have been resolved previously for simplification, which is always possible using simple syntactical transformations and is done on-the-fly by our analysis.[5]

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*

$$(a.x, decl(a)) \in def(a_i) \wedge (b.x, decl(b)) \in (def(a_j) \cup use(a_j)))$$
$$\Rightarrow objects(a, decl(a)) \cap objects(b, decl(b)) \neq \emptyset$$

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

$$formals(a_i) \not\equiv actuals(\texttt{proceed}) \vee$$
$$formals(a_i) \cap def(a_i) \neq \emptyset \vee$$
a_i returns a different value than `proceed`*$(\dots)$.*

[5]For example we transform $x = a.b.c$ to $\$1 = a.b; x = \$1.c$ or $f(g(x))$ to $\$1 = g(x); f(\$1)$ where $1 is a new auxiliary variable.

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. Such advice can easily modify or completely redefine the actual parameters of the `proceed`-call, thus changing values reaching e.g. a called method. Similarly it can also access and modify the return value. The second property explicitly captures these cases.

Additionally to data flow between advice we also have to examine the *control flow*. 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[6] 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.*

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.*

Note that advice interference is restricted by the advice kind. The order of `before` and `after` advice is trivially determined. However, if one of the two pieces of advice is `around`-advice or both pieces of advice are of the same kind, then advice precedence may be undefined and in this case can be picked arbitrarily by the compiler, potentially affecting program semantics. 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 we removed the* `declare precedence` *statement, precedence of these two aspects is undeclared. These two*

[6]Note that we consider explicitly thrown exceptions only. `RuntimeExceptions` due to programming errors (e.g. `NullPointerExceptions`) are ignored in this context.

pieces of advice are relevant *as they are both bound to joinpoints selected by pointcut* `endTiming`*, are of the same kind (both* `after`*-advice) and are defined in different aspects,* `Timing` *and* `Billing` *respectively.*

As the reader may verify, both a_1 *and* a_2 *access the same* `Timer`*-object* o_{tim} *associated 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.*

Note that this criterion, if it succeeds, does not state that the two aspects are independent of each other in general. There might of course be data flow between them. However such data flow does not depend on advice precedence as long as both pieces of advice do not apply at the same joinpoint. Joinpoints are reached subsequently as program execution proceeds,[7] so advice execution order is determined by program control flow. If aspect precedence however is relevant for program semantics, definition 3.6 provides a sufficient criterion to check if an order has to be established for two given pieces of advice.

Note further that although detecting interference among pieces of advice can help considerably to avoid problems, this will not prevent programmers to add two *semantically incompatible pieces of advice* to a system. Consider for example a tracing and an encryption aspect (taken from [5]). The tracing aspect is logging relevant data to be able to easily understand system failures, while the encryption aspect's job is to assert that no data leaves the system unencrypted. Obviously there is a conflict here – we end up with an unencrypted log if logging has a higher precedence than encryption (thus breaking the encryption aspect) or with a encrypted log (hampering logging). Both solutions are not satisfactory. However, if both aspects access common joinpoints, our analysis will at least detect the conflict (as the same data is accessed by both aspects and the encrypting aspects modifies it). If no common joinpoints exist, the system will at least always show the same behavior and not depend on the compiler used, easing debugging in this case.

4 Checking For Aspect Conflicts

In this section we discuss how our criterion has been implemented. Unfortunately not all necessary information can be calculated statically, thus we had to both approximate unknown information and use heuristics.

The first step in our analysis is to find relevant pieces of advice, by analyzing the advice mapping information accessible from the aspect weaver. To check our interference criterion we then examine control dependences and data flow for each pair of relevant advice.

4.1 Basic Data Structures

To implement our analysis we need two basic data structures: the intra-procedural control flow graphs and the advice-aware call graph.

Advice-Aware Call Graph Construction

Call graphs describe the calling relations among methods–and in this context also advice–for a program. Each method and piece of advice is modeled as a node, and an edge from a node n_1 to a node n_2 indicates that a call site or joinpoint, respectively, in n_1 (potentially) invokes n_2. Creating a call graph for a language like AspectJ raises two important issues. First, dynamic binding has to be approximated and second implicit "advice calls" have to be added in the call graph to model joinpoints where advice applies. While the first problem is a standard problem for object-oriented languages [16], modeling of advice application is more interesting. Here we insert explicit call edges from the method containing an adapted joinpoint to attached pieces of advice. While this approach is straightforward for `before` and `after`-advice, `around`-advice is more challenging, as it actually *wraps* the adapted joinpoint, which is only reached if `proceed` is called. Several pieces of `around`-advice thus result in a hierarchy of wrappers, their order implied by advice precedence.

Creating this wrapper hierarchy however faces two problems. First, advice *precedence can be undefined* and second advice application itself can be uncertain due to *dynamic pointcuts*, i.e. pointcut definitions which depend on runtime values and thus cannot be decided statically. To deal with both problems we explicitly model the respective adapted joinpoint and add edges from the containing method to its joinpoint and assume that each advice is called from the adapted joinpoint in turn, thus flattening any wrapper hierarchy to avoid a combinatorial explosion which would result if all possible precedence orders had to be considered.

Example 4.1 (Call Graph Construction) *Consider the simple program shown in figure 3. Sub-figure (a) shows the call graph if we know that* `A1` *has higher precedence than* `A2`*. As we apply* `around`*-advice, the original joinpoint – although not modeled explicitly in this call graph – is replaced by the "call" to the* `around` *advice in* `A1`*. The* `around`*-advice in* `A2` *is called next, as it has*

[7] Note that we do not explicitly handle multi-threaded programs here. Synchronization in this case is similar to synchronization of traditional Java code.

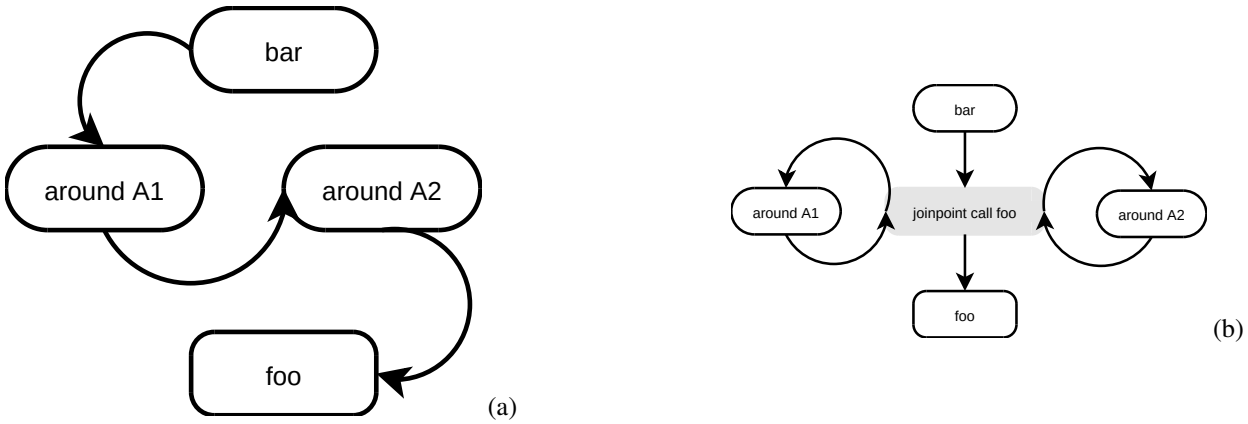


Figure 3. Call Graph Modeling for known (a) and unknown (b) advice precedence.

lower precedence than the previous advice. We thus add a respective edge to the call graph.

However if either aspect application is uncertain due to dynamic joinpoints or advice order is unknown, this modeling of the call graph is incorrect, as paths which actually occur are potentially lost (just assume the opposite precedence order than shown in figure 3 (a)). To deal with this problem, we explicitly model the joinpoint as a special node in the call graph in these cases, as shown in figure 3 (b). Advice applied at this joinpoint is now attached to this joinpoint node, and a call to `proceed` *in the* `around`*-advice links back to this joinpoint instead of an advice or method node.*

With this construction, all feasible paths are represented in the call graph, no paths are lost, however at the cost that now there are also infeasible paths. As this construction is only necessary if precedence order is undeclared the additional imprecision can be tolerated.

Exception-Aware Control Flow Graphs

For each method and each piece of advice we create the control flow graph. In this data structure each statement is represented by a node, and an edge between two nodes indicates potential control flow from one node to the other. This is a standard data structure for static program analysis; creating control flow graphs for advice does not add any additional issues.

However, our analysis also needs information about exceptional control flow. Therefore we augment the control flow graph with this information. While handling of `catch` and `throw` statements is straightforward, method calls (and as well advice application) are more complex as here potentially exceptions thrown in the control flow of the called method (applied advice) can be propagated as well. Therefore we have to calculate the set of exceptions potentially propagated by each method and piece of advice.

Definition 4.1 (Propagated Exceptions) *Let N_t and A_t be defined as before, m be a method or a piece of advice, and e an exception. Then the set of exceptions potentially propagated by m $prop(m)$ is defined as follows:*

$$prop(m) = \{e | \texttt{throw e} \in m\} \cup \bigcup_{n_t \in N_t} prop(n_t) \cup \bigcup_{a_t \in A_t} prop(a_t) - \{e | e \text{ is handled in } m\}$$

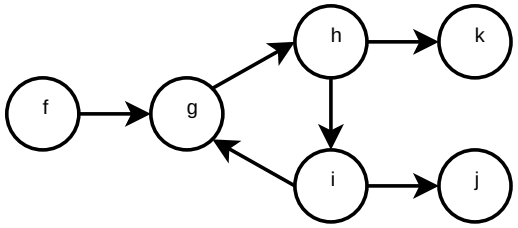
To calculate this information we process the control flow graphs for each method in the call graph in topological order and calculate the set of *propagated exceptions* for each method. For library methods we rely on their `throws`-declarations. Methods not calling any other methods or only library methods are thus the base case of the above recursion. The resulting propagation set is then inserted at each call site to of an already processed method.

To deal with cycles we use a standard technique. First we calculate the set of propagated exceptions for all non-cycle methods which are called by methods within the cycle. Second, we propagate exception sets around the cycle until a fixpoint is reached. These final sets are then propagated to all methods calling cycle members. Once this analysis is finished we have an exception aware control flow graph, i.e. for each method and piece of advice we know the set of exceptions thrown by them.

Example 4.2 *As the* `Telecom`*-example does not contain any recursive methods we use an artificial example to illustrate this approach. Consider the program shown in figure 4. A valid topological order for the call graph shown in (b) is $\{j, k, \{g, h, i\}, f\}$, where nodes $\{g, h, i\}$ form a cycle and thus are treated as a single node for the topological order. We start our analysis with methods j and k, which each throw an exception,* `E1` *and* `E2`*, respectively. As both methods have no call sites and do not handle their exceptions, $prop(j) = \{\texttt{E1}\}$ and $prop(k) = \{\texttt{E2}\}$.*

```
class C {
  void f() { g(); }
  void g() {
    try { h(); }
    catch (E1 e) {}
  }
  void h() { i(); k(); }
  void i() { j(); g(); }
  void j() { throw new E1(); }
  void k() { throw new E2(); }
}
```

(a)



(b)

Figure 4. Calculating the set of Propagated Exceptions.

This information is now propagated to i and h, respectively. However as both methods are part of the cycle, we have to start a fixpoint iteration. We start with $prop(i) = \{E1\}$, due to the call to j. For $prop(h) = \{E1, E2\}$ due to the calls to i and k. Next, we establish $prop(g) = \{E1, E2\} - \{E1\} = \{E2\}$, due to the call to h and the handler for E1. Further propagating this information in the cycle does not further change the propagation sets, thus the fixpoint is reached. So we can propagate the final set for g to establish $prop(f) = \{E2\}$.

Note that the calculated propagation sets do not represent a conservative solution (our analysis misses undeclared `RuntimeException` thrown by libraries as well as exceptions like `NullPointerExceptions` potentially thrown by the virtual machine). We believe that this approach has three important advantages: (i) if we follow the Java convention that `RuntimeExceptions` are programming errors then these exceptions should not be caught but fixed and program semantics (as it should be!) consequently do not depend on advice precedence (crash for any advice order), (ii) we reduce the amount of false positives as each non-trivial piece of advice can potentially throw some `RuntimeException`, and (iii) this simpler heuristic approach is feasible and considerably faster compared to more precise approaches, which is an important property as we envision use of our analysis during compilation.

4.2 Checking Control Dependences

We use the control flow graphs to examine advice control dependence. Note that the way how `before` and `after` advice can affect control flow in AspectJ is rather limited. The only way to do this is by throwing an exception. `Around`-advice however has to explicitly call `proceed`; otherwise the original joinpoint–and lower precedence advice–is not executed. We use the exception-aware control flow graphs to check both properties.

Analyzing `proceed`

To analyze if `around`-advice indeed calls `proceed`, we check if *exactly one* call to `proceed` is *on every path* in the *control flow graph* through the advice using a modified depth first search, which counts the number of `proceed`-nodes visited for each path. If the depth first search reaches an exit statement, the `proceed`-count has to be exactly one in each case. If an already visited node is hit again, the counter must never exceed 1. If this is not the case we report a potential control flow dependence.

Note that due to the heuristic exception analysis this is only a heuristic as well. However we believe that creating a safe analysis producing too many false positives is less valuable than an analysis missing some cases but in general reporting actual problems, especially if missed problems are closely related to programming errors (`RuntimeExceptions`) which should be detected by unit tests and corrected afterward.

Analyzing Exceptions

Analysis of exceptional behavior of advice is straightforward once the propagation sets have been calculated. For a piece of `before` or `after`-advice a we can check our criterion by checking that $prop(a) = \emptyset$.

For `around`-advice however the advice must not change the exception throwing behavior of the call to `proceed`. Thus we have to check that no exception potentially thrown by `proceed` is handled by the `around`-advice and that the advice code throws no additional exceptions. However this information is captured in the exception aware control flow graphs and thus easy to derive from them.

4.3 Checking Data Dependences

To check the data interference criterion we have to resolve aliasing in order to approximate *objects*. Therefore pointer analysis is a suitable technique. As for AspectJ no source level pointer analysis had been available, we implemented our own analysis based on the BDDB-DDB system[17]. Handling of plain Java constructs is well known, so we focus on handling of AOP constructs in this

section. The analysis we implemented for our experiments is both flow and context insensitive (but object sensitive) and is thus rather imprecise, but fast.

Modeling *inter type declarations* is straightforward. Inter type members are visible only in the context of the aspect if declared `private`; otherwise they act as normal class members with one important difference: members of the declaring aspect are also accessible in introduced code. We thus modeled inter type members similar to regular target class members but adapted lookup rules accordingly.

Handling *advice* is more complex. We modeled advice similar to methods, however two important properties of advice have to be considered. First, we have to provide a mapping from exposed *joinpoint context* to formal advice parameters and, second, naturally there is no explicit *advice call*. Advice is implicitly applied at adapted joinpoints. However, as we know the relevant joinpoints (due to the advice joinpoint mapping from the weaver), we know where virtual "advice calls" have to be inserted.

To provide the context mapping, we have to analyze what part of the joinpoint context has been made available to advice via the pointcut. Analysis of the pointcut declaration allows to identify respective variables. AspectJ offers three constructs to explicitly expose context to advice: `this`, `target` and `args`. Besides these, `after`-advice can also give access to return values and thrown exception objects. Determining the variable and thus the object referred to by these constructs depends on the nature of the joinpoint, the AspectJ manual gives a detailed overview. A simple syntactic analysis can provide this information. Once these objects have been identified, we handle "advice calls" similar to method calls by assigning these objects to formal advice parameters and also creating a respective assignment for return values of `around`-advice.

While modifications of heap objects by advice are directly captured by our analysis, `around`-advice can additionally *reassign* parameter values. As a consequence the constraints generated to model the parameter passing potentially depend on the actual advice order.

Example 4.3 (Relevance of Precedence Order) *Figure 5 illustrates this problem. Assuming `A1` has higher precedence than `A2`, parameters are assigned as follows: (i) a = l; b = m; (actuals `foo` → `around` in `A1`), (ii) u = b; v = a; (`proceed` in `around/A1` → `around/A2`) and (iii) x = v; y = v; (`proceed` in `around/A2` → formals `foo`). The opposite order yields the following assignments: (i) u = l; v = m; (actuals `foo` → `around/A2`, (ii) a = v; b = v; (`proceed` in `around/A2` → `around/A1`) and x = b; y = a; (`proceed` in `around/A1` → formals `foo`).*

Figure 5 shows the so called points-to graphs illustrating the data flow in both cases: solid arrows denote an assignment if `A1` has higher precedence than `A2`, dotted lines the other case. An arrow $a \rightarrow b$ indicates that a can point to any object b also points to. We assume that `l` and `m` are directly resolvable to the creation sites shown as rectangles and labeled accordingly. Evaluating the resulting constraints yields the points-to sets shown in figure 5 below the graph. As points-to sets differ *depending on the execution order, we potentially miss conflicts in the interference analysis, if only one order is analyzed.*

Dynamic pointcuts raise a similar issue. AspectJ offers language constructs to restrict joinpoints matched by a pointcut. The keyword `if` for example allows to restrict joinpoint matching based on program values, `cflow` based on the shape of the call stack. Pointcuts containing these constructs cannot be evaluated statically in general. In this setup, simply assuming that advice is applied is not sufficient, as in this case some constraints might be lost if advice actually does not apply, similar to the above case dealing with advice order.

A solution would be to create constraints for each possible order of advice and union the results. However, for n pieces of advice $n!$ different orders exists. If we also consider all subsets due to dynamic joinpoint matching this number even increases. This combinatorial explosion clearly demands a different solution. To avoid it we use a simple trick. Instead of only generating one assignment from actual to formal parameters (following Andersen[1]), we also generate the opposite assignment to identify both points-to sets (as suggested be Steensgard[13]). While this approach is more imprecise it allows to conservatively approximate all possible assignment orders.

With these models for advice and inter type declarations for both the pointer analysis and the call graph it is now possible to derive necessary constraints from the AspectJ source code. We use these constraints together with the call graph as inputs for the BDDBDDB system [17], which uses binary decision diagrams to efficiently solve them.

4.4 Implementation and Example

Implementing the interference criterion based on the results of the pointer analysis and the exception-aware control flow graphs is now straightforward. If we are only interested in a binary information (interference or not), we can abort the analysis for two pieces of advice once a criterion violation is found. However in general it is interesting for programmers to know *why* two pieces of advice interfere. We thus continue with the analysis to collect all objects and access patterns where the two pieces of advice potentially conflict. Thus even if our analysis reports false positives, the programmer has more detailed information to make a well-informed decision if or how precedence for two aspects has to be declared.

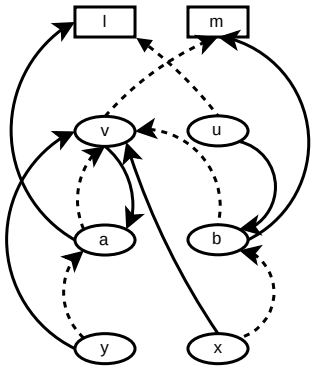
We implemented our analysis as a set of plug-ins for the Eclipse IDE. Our system correctly identifies the data

```
class SomeClass {
  void zip()    {
    ...
    foo(l, m); }

  void foo(Object x,
           Object y) {
    ...
  }}
```

```
aspect A1 {
  around(Object a,
         Object b):
    call(foo(..))
      && args(a, b){
    ...
    // switched!
    proceed(b, a);
    ...
  }}
```

```
aspect A2 {
  around(Object u,
         Object v):
    call(foo(..))
      && args(u, v) {
    ...
    // only forward v!
    proceed(v, v);
    ...
  }}
```



Variable	a	b	u	v	x	y
A1 before A2	{l}	{m}	{m}	{l}	{l}	{l}
A1 before A2	{m}	{m}	{l}	{m}	{m}	{m}

Figure 5. Generated constraints depend on advice order.

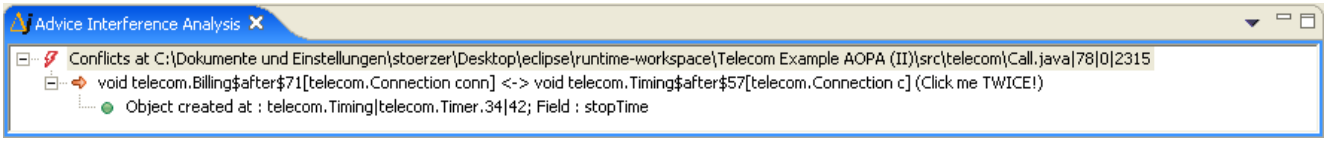


Figure 6. Interference Analysis Results for `Telecom`

flow conflict if the `declare precedence` statement is removed in the `Telecom` example. The pointer analysis reveals that both `Timing` and `Billing` access the same `Timer` objects, where `Timing` sets `stopTime` and `Billing` reads this value to finally bill the call. As neither piece of advice throws a (checked or declared) exception, no control flow interference is found here.

Figure 6 shows a screen shot of the results presented to the user. Clicking on the first line of the view opens the editor and presents the joinpoint affected by both pieces of advice to the user. Clicking on the second line in turn opens either the `Timing` or the `Billing` advice and the final line allows to access the creation site of the timer object, so allowing the user to quickly examine the analysis results.

If aspect order is defined by adding the `declare precedence` statement, this is also detected by our system and no warnings are generated. Clearly our example can only give a first impression of the effectiveness of our approach and additional case studies are needed to better evaluate it. However this is hampered by the fact that unfortunately only few AspectJ programs are publicly available. However some programs are available, and additional case studies are planned.

Although experience with our prototype is limited, we expect our system to scale to at least medium size programs as we use an efficient system for the most expensive part of the analysis–the pointer analysis. BDDBDDB claims to provide this scalability. For medium size AO systems maintaining an overview over all applying aspects and their peculiarities is already hard, and thus applying our system to capture precedence related problems is valuable.

The precision of our analysis could clearly benefit from improvements in the pointer analysis underlying our algorithm, especially by using a context-sensitive pointer analysis. Future work will on one hand increase precision for our analysis by switching to such an analysis. Second, we are currently in the process of refactoring open source Java systems to generate some subjects to better evaluate our system and also generate some interesting runtime data.[8] To further increase precision and reduce spurious data dependence warnings, we could also try to resolve dynamic joinpoints statically as far as possible. As this is also an important topic for AspectJ performance optimization [2] and program analysis research in general we consider this to be orthogonal to our work.

5 Related Work and Conclusions

A shortened version of this work has been published in [14]. In general our work is related to pointer analysis and aspect interference analysis. We start with a very short overview of pointer analysis in literature which is necessarily not comprehensive, as a large body of work on this topic exists.

Andersen presented a subset-based algorithm for pointer analysis for the C language in [1], which is basically also used in this work. As this algorithm is relatively expensive ($O(n^3)$), Steensgard [13] proposed a simpler algorithm which identifies points-to sets on assignments, resulting in almost linear runtime, however also a considerable loss of

[8] For the `Telecom` example runtime is just few seconds.

precision. Extensions for object-oriented languages have to deal with dynamic binding. Due to space restrictions we refer to [7] for an overview of available algorithms.

Although the problem of aspect interference has been recognized by researchers, there are still only few approaches in literature addressing this problem. In [4] the aspect interaction problem is discussed in a position paper, although on a more abstract level and targeted to the composition filter approach [3]. While this work contains an interesting discussion of problem itself, a solution is only briefly outlined. In [11] a reflective aspect-oriented framework is proposed which allows users to visually specify aspect-base and aspect-aspect dependences using the Alpheus tool. The tool is also used to specify aspect conflicts and resolution rules which are then resolved automatically at runtime. While the framework offers a more abstract view on aspects and provides a richer set of conflict resolution rules than AspectJ (thus leveraging some of the problems discussed in this paper), conflict detection is manual.

In [5, 6], 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 that 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 [12] 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, our work differs from their's in several ways. First, we apply the analysis to determine effects of aspect-aspect rather than aspect-base interference. Second, their algorithm–while more precise–is also considerably more expensive than our BDDBDDB-based pointer analysis, thus potentially allowing to analyze larger systems. As our analysis focuses on joinpoints with multiple pieces of applied advice, loss in precision seems acceptable in favor of gained performance.

The problem that conflicts are not reported at all has also been reported as a bug for AspectJ, and a compiler warning has been suggested to deal with this problem. While this makes programmers aware of potentially conflicting advice, our analysis is able to detect commutative advice thus providing more precise feedback.

Finally, in [8] Ishio et. al also address the increased complexity of AspectJ. To support program debugging they propose two techniques: First, they analyze call graphs and implemented a tool to automatically detect potential infinite recursion due to careless pointcut design. Second, they calculate dynamic slices based on a technique called DC slicing to help programmers isolate failures in code. To create the underlying data structures, they discuss similar problems as discussed here and also in part propose similar solutions, although application of their work is completely different.

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 used standard program analysis techniques to implement this criterion and discussed the results of our implementation for the `Telecom` example.

Our approach can support programmers working with AO systems who have to deal with the advice precedence problem by helping to avoid unexpected side effects during system contruction and evolution.

Acknowledgements Thanks to Daniel Wasserrab for comments on the draft version of this technical report and to the anonymous reviewers of our earlier ASE paper, whose feedback also helped to improve this extended technical report.

References

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. DIKU report 94/19.

[2] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhotàk, O. Lhotàk, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Optimising Aspectj. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 117–128, New York, NY, USA, 2005. ACM Press.

[3] L. Bergmans and M. Aksits. Composing crosscutting concerns using composition filters. *CACM*, 44(10):51–57, 2001.

[4] 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.

[5] 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.

[6] 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.

[7] M. Hind and A. Pioli. Which pointer analysis should i use? In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 113–123, New York, NY, USA, 2000. ACM Press.

[8] T. Ishio, S. Kusomoto, and K. Inoue. Debugging Support for Aspect-Oriented Program Based on Program Slicing and Call Graph. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on Software Maintenance*, Washington, DC, USA, 2004. IEEE Computer Society.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the Twenty-Third ACM Symposium on Principles of Programming Languages*, pages 32–41, St. Petersburg, FL, January 1996.

[14] M. Stoerzer, R. Sterr, and F. Forster. Detecting precedence-related advice interference. In *In Proceedings of 21th International Conference on Automated Software Engineering (ASE)*. IEEE Press, September 2006.

[15] P. Tarr and H. Ossher. Hyper/j: multi-dimensional separation of concerns for java. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 729–730, Washington, DC, USA, 2001. IEEE Computer Society.

[16] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 281–293, New York, NY, USA, 2000. ACM Press.

[17] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 131–144, New York, NY, USA, 2004. ACM Press.