# Detecting and Resolving Ambiguities caused by Inter-dependent Introductions

Wilke Havinga, Istvan Nagy, Lodewijk Bergmans & Mehmet Aksit
University of Twente, The Netherlands
{havingaw,nagyist,bergmans,aksit}@ewi.utwente.nl

## ABSTRACT

AOP languages are continuously evolving, for example (1) pointcut languages are becoming increasingly powerful with respect to the expressiveness of the pointcut language itself, (2) new program properties can be used as a selection criterion in pointcut designators, or (3) new types of program elements can be introduced by means of a crosscut specification. In this paper we investigate the consequences of these trends. To this end, we focus particularly on the usage of meta-data annotations: several recent (versions of) AOP languages support the use of annotations as a selection criterion in pointcut designators or the introduction of annotations, or both. We investigate the introduction of annotations through the use of expressive pointcut languages; explain why introduction of annotations is useful, and in particular, why and how annotations can be derived from other annotations.

We explore the issues that arise due to the inter-dependencies between annotation introductions. We investigate when such dependencies may cause ambiguities, and we present an algorithm that resolves the dependencies when possible, and detects ambiguous cases that cannot be resolved. The solution we propose is implemented within the *Compose** tool, which supports the introduction of meta-data annotations.

## 1. INTRODUCTION

A trend in aspect-oriented programming (AOP) is the development of more expressive pointcut languages. One way to increase the expressiveness of pointcut languages is by adding support for metadata annotations—a mechanism that enables the attachment of meta-data to program elements. Various object-oriented programming languages (such as C# [7] and Java [16]) now support such annotations. The annotations do not have any direct influence on the execution of an application, but they can be used by compile-time tools or meta-facilities. AOP and annotations are a natural match, as annotations can be used to explicitly add *design information* to a program. Without annotations such information is usually implicit, buried within an implementation. Explicit design information can be used to write pointcuts that are much less fragile than those that are based on e.g. naming conventions or structural patterns [15].

In this paper, we investigate the use of introductions in aspect-oriented programming, especially focusing on the use of annotations. We describe how an expressive pointcut language can be used to facilitate the *introduction* and *derivation* of annotations. We identify ambiguity problems that can be caused by using expressive pointcut languages combined with the introduction of elements (e.g. annotations) in the same structure that is queried by the pointcut language. We perform an extensive analysis and discuss approaches that can solve these problems. We present a solution within the context of a concrete implementation of the *Compose** language [4, 6]. Finally, we argue that these problem are inherent to the combination of expressive pointcut languages and introductions, and are not limited to only specific tools or the use of annotations. The main contributions of this paper are the identification and analysis of ambiguity problems related to introductions, as well as a solution that can detect and/or resolve these problems.

## 2. USING ANNOTATIONS IN AOP

In this section, we explain how annotations can be used in pointcuts. Next, we describe how annotations can also be introduced and derived through the use of pointcuts, and explain why this is useful.

### 2.1 Using annotations in pointcuts

Pointcuts that directly refer to the structure or syntax of a program are generally quite fragile: they often break when the base program is refactored. To avoid such fragile pointcuts, we can decorate the program with annotations that explicitly represent design information about the program. These annotations can then be used as a selection criterion in pointcuts. We demonstrate this use of annotations by giving a small example: listing 1 shows a simple Java class.

```
1 @PersistentRoot class User {
2   String name;
3   String email;
4   SessionID session;
5   ..
6 }
```

**Listing 1: Annotations in Java**

*AOSD 06,* March 20–24, 2006, Bonn, Germany

The notation *@PersistentRoot class User* specifies that the class *User* has the annotation *PersistentRoot* attached. Annotations can be attached to several kinds of program elements, most notably classes, methods and fields.

The following AspectJ (version 5) example shows a pointcut that selects the execution of all methods within classes that have the annotation *PersistentRoot*:

```
1   pointcut isPersistent():
2      execution(* (@PersistentRoot *).*(..));
```

**Listing 2: An annotation-based pointcut**

Within the *execution* part of this pointcut, the first * indicates that any (method) return type will match. The next part, *(@PersistentRoot *)*, means the class containing the executed method must have the annotation *PersistentRoot* attached. The remaining part specifies that methods with arbitrary name and (number of) parameters will match.

By writing pointcuts based on annotations, we can select program elements based on design information, rather than relying on naming or structural patterns. Because the design intention is made explicit, such pointcuts can be more robust [15].

However, for some cases the usefulness of annotations can still be improved. For example, by directly binding the annotation *PersistentRoot* to every persistent class, those annotations are scattered over the application. In fact, we can state that the annotation *PersistentRoot* is therefore tangled with base classes (such as the class *User* in this example), because their code cannot be fully separated. This may hinder the reusability and evolvability of the classes involved.

For this reason, it may be desired to make annotations application specific: suppose the class *User* above would be reused in two applications. In one application, it is part of a set of classes that should be made persistent. Another application has a different set of persistent classes, which does not include the class *User*. To work around this issue, the applications would have to specify differently named annotations in the reusable base classes (e.g. *App1Persistent* and *App2Persistent*), thus 'polluting' an otherwise reusable class with application-specific information. Also, third party tools that depend on the use of certain annotations (with a fixed name) cannot be used in such a situation.

On the other hand, it is important to note that in certain domains (e.g. security) programmers may intentionally want to bind annotations statically (in the source code) to ensure that a (fixed) set of constraints is met by all applications reusing that particular module. In such cases, every application that reuses a program element with annotations will have the same set of annotations.

Fortunately, recent (versions of) AOP languages support both types of scenarios: they support the use of statically bound annotations (those that are directly attached to the source code) as well as *introduced* annotations. In the latter case, a crosscutting specification is used to indicate where annotations should be attached. The next section discusses this technique.

## 2.2 Introduction of annotations

An intuitive AOP solution to the problem of scattered annotations is to *introduce* annotations (from within an aspect) on program elements selected by a pointcut. Note that the language used to designate places where annotations have to be introduced can be different from the normal pointcut language[1]. In practice, the expressiveness is usually similar to that of the normal pointcut language - and we show that this expressive power is indeed desired, as it enables programmers to express their intented designs more directly. In AspectJ, the introduction of annotations can be expressed as follows:

```
declare @type : SessionID+ : @TransientClass();
```

**Listing 3: Introducing an annotation**

This example specifies that the class *SessionID* and all its subclasses should be marked with the annotation *TransientClass*. This annotation expresses some information about the design: presumably, in this application, it does not make sense to permanently store session-ID's, as they represent volatile information that expires after a limited amount of time. By explicitly specifying this information separate from the base classes, the information is localized in one place. The AspectJ construct *declare @type* means 'introduce an annotation on a type'. The part between the colons is a standard AspectJ Type-pattern [2], in this case matching the type *SessionID* and its subclasses. The part after the second colon specifies that the annotation *TransientClass* should be attached to the types that match the specified pattern.

It should be noted that the pattern expression used to introduce annotations does not have full expression power: in this case, we can only express restrictions directly related to the type that we want to attach an annotation to. Still, this allows us to express conditions on the introduction of annotations. As these condition-patterns can include restrictions based on annotations, we can in fact *derive* the existence of one annotation based on the existence (or absence) of others. The next subsection explains how this can be useful.

## 2.3 Deriving annotations

Annotations may be relatively bound to each other: an annotation can be attached to a certain program element, if another —related— program element has a certain annotation.

To illustrate this, we extend the previous example by defining the following rule: If a field is of a type that has the annotation *TransientClass*, it should be marked by the annotation *TransientField*. This way, we can specify an exception to the general rule that all fields within a class marked by the annotation *PersistentRoot* will be kept in a persistent datastore.

In AspectJ, it is possible to express such a rule as follows:

```
declare @field: (@TransientClass *) * :@TransientField();
```

**Listing 4: Deriving an annotation**

[1]In AspectJ terminology, the selection part is called *pattern*. We use a more general notion of the term *pointcut* to also indicate such selection patterns.

This declaration specifies that the annotation *TransientField* should be introduced on all fields that match the field-pattern between the colons. Here, the pattern specifies that such fields should be of a type that has the annotation *TransientClass* attached, and that we do not put any constraints on the name of the type or the name of the field. As you can see, the introduction of the annotation *TransientField* *depends on* the fact whether the annotation *TransientClass* is attached to the type of a field.

Another example can be found in [5], where the following *constraint* is specified and enforced: "If a class is decorated with the annotation *WebService*, its public methods (that constitute the webservice) *should have* the annotation *WebMethod*". The difference with the approach we discuss here is that we actively *generate* the dependent annotations, i.e. "If a class is decorated with the annotation *WebService*, its public methods *are marked* with the annotation *WebMethod*". As such rules can express the intention of a programmer more directly, the ability to express such rules has a clear added value.

One might ask why it is useful or necessary to introduce the annotations *TransientField* or *TransientClass* if their places can also be designated directly by the pointcuts that could express the rules above. Using such pointcuts can in fact be sufficient when these annotations are only used within pointcut expressions of aspects. However, (derived) annotations can be used by third party tools or frameworks as well. In many cases, we can *derive* whether a certain annotation should be attached based on the existence of other annotations, certain types of statements or structural combinations of program elements (i.e. 'software patterns'). In these cases, using derivation removes the need to manually specify where annotations have to be attached (either in the concern source or the source of the base application).

# 3. PROBLEM STATEMENT

In the previous section we described how annotations can be used in pointcuts. We also saw that annotations can be *introduced* through the use of pointcuts. In cases where such pointcuts refer to other annotations, we speak of *deriving* annotations, based on the existence or absence of other annotations. We have shown that such derivation rules are useful, because they give the programmer a more direct way to express design intentions. However, basing introductions on derived information in this way may result in dependencies between introductions (e.g. of annotations) and the pointcuts used to designate places where introductions should be applied. This section describes the problems that such dependencies may cause.

The main problem caused by dependencies between introductions of annotations is that the order of applying the introductions may matter. If this order is unspecified, ambiguities may occur. For instance, consider the combination of the examples in listings 3 and 4:

```
1 declare @type : SessionID+ : @TransientClass();
2 declare @field: (@TransientClass *) * :@TransientField();
```

**Listing 5: Inter-dependent declarations**

Both declarations (see listing 5) specify the introduction of an annotation; one has conditions based on the structure (line 1), while the other depends on the existence of another annotation (line 2). We assume that, without further specification, there is no ordering implied between these two introductions (they could even be part of two different aspects). In that case, there are two possible orderings of applying these introductions. The first is to start by evaluating the types matching the condition *SessionId+* and attaching the annotation *TransientClass* to these types (i.e. apply the first introduction), and then to apply the second introduction, which attaches the annotation *TransientField* to fields of a type that have the annotation *TransientClass* attached. The other possibility is to switch the order of applying the two introductions. In that case the annotation *TransientField* will not be introduced anywhere, as the annotation *TransientClass*, on which this introduction clearly depends, has not been attached yet. In this case, it is intuitively clear which order is intended by the programmer: the first order is desired, as the declaration in listing 4 clearly depends on the one in listing 3. Thus, it makes sense to apply the introduction in listing 3 first.

However, such a conclusion cannot always be drawn. Depending on the expressiveness of the pointcut language used to specify where annotations should be introduced, it may not be trivial to see which dependencies exist. For example, when using a Turing-complete pointcut language, it is generally impossible to infer such information by analyzing only the textual representation of pointcut expressions. Using a Turing-complete pointcut language to specify pointcuts can be useful, as it allows for powerful reasoning within pointcut expressions [10]. However, even when a more restricted pointcut language is used, there are still several issues to be addressed.

For one thing, there can exist multiple levels of dependencies. This can already be seen in the example above, as the attachment of the annotation *TransientClass* depends on the (static) structure, while at the next level, the attachment of the annotation *TransientField* depends on the attachment of the annotation *TransientClass*. In general, there can be 'chains' of introductions that depend on each other. This suggests some kind of iterative algorithm is needed to resolve the dependencies between introductions of annotations, as it may not be possible to resolve all dependencies in a single pass over all the introductions. However, dependencies between introductions can even be circular. If circular dependencies occur, such an algorithm might never terminate.

In addition to this problem, the existence of dependencies implies that the ordering of evaluating the declarations can lead to different results, as seen in the example above. In the example it is intuitively clear which ordering is desired. However, given the occurrence of circular dependencies and/or negative dependencies (i.e. where an annotation should only be attached if another annotation is *absent*), there is no intuitive way to tell which order of evaluation is intended by the programmer. In such cases, the specification of annotation-declarations may be ambiguous, unless certain ordering constraints are implied or can be specified by the programmer.

In AspectJ, the ordering of introductions within a single aspect is unspecified. Based on some small experiments, we conclude that the current implementation detects simple de-

pendencies such as in listing 5, but yields ambiguous results (i.e. depending on the ordering in the aspect sources in seemingly arbitrary ways) when negative or circular dependencies are involved. When introductions are divided over several aspects, the ordering is also unspecified. Thus, the current implementation does not address the problems related to ambiguous (non-declarative) specifications and the expression of circular dependencies.

We stress that these types of problems are not necessarily limited to introductions of annotations only. Any type of introduction that changes parts of the structure that can also be used in pointcuts describing other introductions can potentially lead to similar problems. Whether problematic cases can actually occur depends on the expressiveness of the pointcut language. In this paper, we discuss potential solutions to these problems, and discuss the trade-offs that have to be made between supporting expressive pointcut languages combined with introductions on one hand, while on the other hand offering a predictable, intuitive and non-ambiguous programming interface. Also, we want our AOP language compiler to be able to resolve dependencies whenever possible, and to detect cases where the specification is ambiguous.

The remainder of this paper describes how the problems that we observed can be addressed, as illustrated by the aspect-oriented language *Compose**.

# 4. ANNOTATIONS IN COMPOSE*

In this section, we explain the use of annotations within the context of Compose*, our aspect-oriented programming language based on the concept of Composition Filters [3, 8], implemented on the .NET platform. In the Composition Filters paradigm advice is specified by filter modules, which can be superimposed (woven) on any number of classes. The selectors (cf. pointcuts) that specify where filter modules (advices) should be superimposed are written in prolog, using a set of predicates that can query the structure of the program.

## 4.1 Annotation-Based Join Point Selection

Listing 6 shows part of an example concern handling *ObjectPersistence*. The selector *persistentClasses* selects all classes that have the annotation *PersistentRoot* attached, using a predicate-based selector language that can select program elements that are part of the static structure of the application. In this example, the variable *C* selects all program elements, provided that they are classes that have the annotation *A* attached, which (as specified on line 7) must be a program element of the annotation type that is named *PersistentRoot*. The selector *persistentClasses* is thus equivalent to the AspectJ pointcut in listing 2.

```
1
2 concern ObjectPersistence {
3   superimposition {
4     selectors
5       persistentClasses =
6        {C | classHasAnnotation(C, A),
7             isAnnotationWithName(A, 'PersistentRoot')};
8   ...
```

**Listing 6: Using annotations in Compose***

## 4.2 Superimposition of Annotations

We introduce a simple extension to the existing mechanism used to superimpose filtermodules in Compose*: a new language construct that specifies the superimposition of annotations on a set of selected program elements. The selector mechanism itself is exactly the same as the one used for superimposing filtermodules, and thus has Turing-complete expressiveness. Program elements can be selected based on their name, properties and relations to other program elements (i.e. based on the static structure of the application), including annotations.

Listing 7 shows the same example as used for AspectJ in listing 3, now implemented in Compose*:

```
1  concern AppSpecificPersistence {
2    superimposition {
3      selectors
4        transientClasses =
5          { AnySess | isClassWithName(S, 'SessionID'),
6                      inheritsOrSelf(S, AnySess) };
7      annotations
8        transientClasses <- TransientClass;
9    }
10 }
```

**Listing 7: Superimposition of annotations**

In listing 7, the class *SessionID* and its subclasses are selected by the selector *transientClasses* (line 4-6). The annotation *TransientClass* will be *superimposed* (introduced) to this set of selected classes (line 8).

In principle, annotations can be superimposed on any kind of program element represented in the language model used by the pointcut language. In practice, there can be limitations in the underlying implementation framework (in our case, .NET). In addition, annotation types themselves can restrict the target program element types to which they can be applied. So far, we mainly considered the attachment of annotations to classes, methods and fields. Conceptually it would be possible to annotate AOP constructs as well, such as concerns or filter modules. AspectJ does in fact support the use of annotations on e.g. aspects and advice. This topic is discussed in more detail in [15]; section 3.2.1 (regarding the language model) and 5.2 (regarding the use of annotations on AOP-specific program elements).

## 4.3 Derivation of annotations

In the previous sections, we extended the selector language of Compose* to use annotations as a selection criterion and introduced a language construct to superimpose annotations on a set of selected program elements.

These two features can be combined to achieve the derivation of annotations. Again, in listing 8 we show the same example as expressed in AspectJ in listing 4:

```
1 superimposition {
2   selectors
3     transientFields = { F |
4       typeHasAnnotationWithName(T, 'TransientClass'),
5       fieldType(F,T) };
6   annotations
7     transientFields <- TransientField;
8 }
```

**Listing 8: Deriving an annotation**

Here, the selector *transientField* selects all fields *F*, as long as they are of a type that has the annotation named *TransientClass* attached (line 3-5). The annotation *TransientField* is superimposed on these fields (line 7).

There are two big differences between the selector language used to superimpose annotations in Compose* and the pointcut language used to introduce annotations in AspectJ. First, Compose* offers predicates that express all (relevant) properties of and relations between program elements. For example, the predicate *typeHasField(T,F)* expresses the relation between a field and its containing type. Using this relation, we can select e.g. only fields in classes that have the annotation *PersistentRoot* attached, and introduce an annotation on those fields (listing 9 gives an example). In AspectJ we cannot express such a pointcut, as annotation introductions on fields follow the pattern: *declare @field : FieldPattern : @AnnotationToAttach()*. The FieldPattern can place restrictions on only the type and the name of the field itself (see e.g. the example in listing 4), and not on anything else, such as for example the type containing a specific field.

```
1 superimposition {
2   selectors
3     persFields = { F |
4       typeHasAnnotationWithName(T, 'PersistentRoot'),
5       typeHasField(T, F) };
6   annotations
7     persFields <- PersistentField;
8 }
```

**Listing 9: Using relations between program elements**

The second difference is the inherent expression power of the pointcut languages. Compose* uses a Turing-complete general-purpose language (prolog) with a predefined library of predicates, whereas AspectJ uses a more strictly defined, less expressive pointcut language. Basically there is a tradeoff between supporting powerful reasoning *within* pointcut expressions (here, Compose* offers more power) as opposed to reasoning *about* pointcut expressions (which is easier in AspectJ).

Clearly, the problems described in section 3 also apply to our implementation of annotation introductions in Compose*: there can be dependencies between the superimposition of annotations and the selectors used to specify these superimpositions, because these selectors can also be based on annotations. In the next section we analyze these problems.

# 5. PROBLEM ANALYSIS

As discussed in the problem statement, introductions based on expressive pointcut languages can cause problems related to the desired order of applying the introductions. The existence of dependencies between introductions can cause ambiguities, i.e. when different orderings of applying introductions leads to different results.

In this section, we analyze concrete cases where such problems occur, and discuss potential ways to solve these problems.

## 5.1 Detecting dependencies

We want to solve the ordering problems described in the problem statement for the most general case, i.e. for a Turing-complete pointcut language, such as the one used in Compose*. Although such a language allows for powerful reasoning in selector expressions, it makes it impossible to reason reliably about the results of evaluation by looking at the source code of selector expressions. The cause is that an expression in a Turing-complete language is, in general, not statically decidable, i.e. we cannot (always) know the results without evaluating the expression. We give a small example to demonstrate this.

In the context of .NET, annotation types are basically normal types. This means it is also possible to build (inheritance) hierarchies of annotations. For example, we can create a generic annotation type called *PersistentRoot*, and another annotation type *XMLPersistentRoot* that extends *PersistentRoot*. In listing 10, we show how to select all classes that have *any* kind of *PersistentRoot* annotation attached.

```
1 ClassPersistent =
2  { C | isAnnotationWithName(PRAnnot, 'PersistentRoot'),
3        inheritsOrSelf(PRAnnot, AnyPRAnnot),
4        classHasAnnotation(C, AnyPRAnnot) };
```

**Listing 10: (in)visibility of dependencies**

The problem with this selector is that its evaluation may depend on the existence of a number of annotations, i.e. in this case any annotation that extends the annotation *PersistentRoot*. However, by looking at only the source code, we cannot generally predict which annotations will become bound to the free variable *AnyPRAnnot*. Therefore, we cannot infer (from statically analyzing the source) that there exists a dependency between this selector expression and the superimposition of e.g. the annotation *XMLPersistentRoot* somewhere else.

However, we can look at the *results* of selector evaluation; if the result of evaluating a selector changes after we have superimposed a certain annotation, there obviously exists a dependency. The reverse is not true: the fact that the result does not change after superimposing an annotation does not imply that there is no dependency. It just does not occur given the current combination of selectors, program elements and annotations in the application under consideration. In other words, by performing the evaluation of the selectors, we can observe when dependencies occur, but we cannot detect *potential* dependencies that are independent of a particular application.

Hence, our approach to determine a correct order of evaluation is based on trying all possible orders of evaluating the selectors and superimposing annotations, and then observing the results. Such an iterative approach can also solve the problem of multi-level (or even circular) dependencies, as we explain in the next section.

## 5.2 Circular dependencies

Circular dependencies between introductions may occur intentionally. For example, consider the combination of the following two rules (taken from [5]): (1) If a class contains a public method that has the annotation *WebMethod*, the class itself should have the annotation *WebService*, and (2) If a class has the annotation *WebService*, all of its public methods are *WebMethods* that should be annotated accordingly.

Both rules can easily be expressed by annotation introductions in Compose*. However, because these rules depend on annotations that are introduced based on the other rule, iteration over those introductions is required.

This problem can be addressed by iterating over the evaluation of selectors and the introduction of annotations, until a fixpoint is reached. That is, the algorithm iterates over the introduction of annotations until the state (the set of selected program elements per selector) does not change between two iterations. An annotation can only be superimposed once on each program element—if it is attached a second time in a later iteration, the results are considered idempotent. In each iteration step, the superimposition of an annotation is performed and the selectors are reevaluated to reflect the changes caused by the superimposition. Because the resolution is based on the reevaluation of selectors between the iterative application of introductions, we do not need to know where dependencies occur beforehand.

Circular dependencies may also occur unintentionally, for example by combining several aspects that derive annotations from the existence or absence of other annotations. In certain cases such circular dependencies may cause infinite loops within an iterative resolution algorithm. To illustrate such a case, we show an example in listing 11. In this listing, we express the rule that the annotation *PersistentRoot* should be introduced on classes that (1) do not have this annotation[2] and (2) have at least one field that should be made persistent.

```
1 selectors
2   isPersistentRoot = {C |
3     not(classHasAnnotationWithName(C,'PersistentRoot')),
4     classHasField(C,F),
5     fieldHasAnnotationWithName(F, 'PersistentField') };
6 annotations
7   isPersistentRoot <- PersistentRoot;
```

**Listing 11: Example of a circular dependency**

The problem with this rule is that, in an iterative approach, after executing the superimposition of the annotation *PersistentRoot* on classes that matched the selector, we have to reevaluate the selector *isPersistentRoot*. However, after reevaluation it will no longer match the classes on which we just introduced the annotation *PersistentRoot*, because it no longer fulfills the first condition. Because the rule on which the superimposition was based is now violated, a declarative approach would suggest that the superimposition should not have been executed in the first place. However, if we would 'roll back' that decision, the rule would apply again. Hence, we would create an infinite loop caused by what we will call *negative feedback* between selectors and the process of superimposing annotations. In this case it is relatively easy to recognize the problem, as the selector and superimposition are specified in one place; but in practice, the selector and superimposition specification may be distributed over different aspects.

This example demonstrates that we cannot ensure that iteration over selectors and superimposition of annotations will terminate, given the occurrence of *negative feedback* - that is, introductions that depend on other annotations being *absent*, while these annotations are subsequently also introduced (possibly in a completely different location).

However, many pointcut languages do support exclusion operators, as this can be very useful. A good example (that can be easily expressed in both AspectJ and Compose*) would be to specify that fields that do *not* have the annotation *TransientField* automatically get the annotation *PersistentField* superimposed. However, as we will show in the next section, a combination of such rules (even though they can easily be expressed in existing languages) can even lead to specifications that are inherently ambiguous (in the absence of explicit ordering constraints). Obviously, we would like to detect the occurrence of such situations.

## 5.3 Ambiguous selector specifications

The existence of exclusion operators and dependencies in the selector language leads to another problem: we prefer superimposition specifications and selectors to be declarative; their specification should not imply any ordering of superimposition. However, different orders of evaluating the selectors and executing the superimposition of annotations do exist. As a consequence, this may result in different sets of program elements with different annotations attached. If this happens, the concern specification is ambiguous and non-declarative. As an example, consider the ambiguous combination of rules specified in listing 12.

```
1 selectors
2  notTransient = { F |
3   not(fieldHasAnnotationWithName(F, 'TransientField'))};
4  notPersistent = { F |
5   not(fieldHasAnnotationWithName(F, 'PersistentField'))};
6 annotations
7   notTransient <- PersistentField;
8   notPersistent <- TransientField;
```

**Listing 12: Ambiguous selector specification**

Listing 12 lists two rules: if a field is not transient, it should be made persistent, and vice versa. These two derivation rules express the design intention that each field in a program should be marked either transient or persistent. However, the design intention is not expressed *precisely* in this example: suppose there exists a field that has neither annotation *TransientField* nor *PersistentField* attached. This field is then selected by both selectors, *notTransient* and *notPersistent*. If we superimpose the annotation *PersistentField* on it first (line 5) and then reevaluate the selectors, the field no longer matches selector *notPersistent*, so the annotation *TransientField* will not be attached. However, if we first superimpose the annotation *TransientField* (line 6), the field no longer matches the selector *notTransient*, so the annotation *PersistentField* will not be attached. In this case, the end results are different depending on which introduction is executed first. The specification is syntactically valid, but also clearly ambiguous, as there is no way to discern which order of applying the introductions was intended by the programmer.

In any case, we would like to detect such problems, either to forbid the use of such ambiguous specifications, or to let the programmer specify an ordering that would resolve the problem.

[2]Actually, this part of the rule is superfluous (applying the same annotation a second time has no effect), but the code is valid; we use it to demonstrate the type of problems that can occur when programmers specify such rules.

## 5.4 Summary

Based on the observations made in the previous subsections, we draw the following conclusions with respect to the ordering problems:

- The expressiveness of the selector language in Compose* does not allow reasoning about dependencies based on the textual representation of the selectors. In addition, there can be multiple levels of dependencies (including circular dependencies). For these reasons, we apply an approach based on the iterative evaluation of selector expressions and superimposition of annotations.
- Iterative resolution of dependencies will not terminate (i.e. it leads to infinite loops) in cases where a circular dependency occurs in combination with an exclusion operator. We illustrated such a case in section 5.2.
- An important issue is that selectors and superimposition should be declarative, which means that the order of attaching annotations should not matter (otherwise, the specification is imperative rather than declarative). Implying an ordering compromises the declarative nature of selector specifications, which leads to problems regarding evolvability: introducing additional selectors may change the implied ordering. In addition, this makes it harder for programmers to see what is actually selected by a particular selector.

These conclusions form the core of our solution proposal: an algorithm that considers all different orderings in which the annotations can be superimposed, and iterates over every possible ordering, until the set of selected elements for each selector reaches a fixpoint. To address the problem of infinite loops, we disallow the occurrence of negative feedback between selectors and superimposition of annotations. This means that selecting based on the *absence* of an annotation that is attached by another concern will be considered an error, as this causes negative feedback (hence, the case described in listing 11 would be detected as problematic). Note that this does not directly limit the expressiveness of the selector language itself: it is still possible to use 'not' and other types of exclusion constructs. However, it does limit the possibilities of introductions, as it is no longer allowed to introduce annotations that are used to *exclude* program elements in selectors.

If such *negative feedback* occurs, the programmer has two options to make the specification declarative (apart from changing the design): (1) do not derive annotations that are used as part of exclusion conditions in selectors, but attach them manually (in the source), or (2) do not base the exclusion condition in the selector on a derived annotation, but directly on the (static) conditions that where also used to specify the derivation of the annotation. Both solutions are workarounds that do not yield very elegant code, but can at least always resolve the issue. The discussion section explains why we think the alternative (i.e. specifying an ordering to resolve the problem) is less attractive.

Additionally, we disallow ambiguous specifications by considering the occurrence of different results based on different orders of attaching the annotations as an error. To detect such cases, our algorithm tries every possible ordering of introducing the annotations.

In fact, we observed that the second restriction we impose on the use of introductions (i.e. different orderings are not allowed to render different endresults) may be superfluous, because we observed that by disallowing negative feedback, the order of superimposing the annotation can never lead to different end results. In other words, cases that would be detected as being an ambiguous specification would always involve negative feedback (and already be detected as such). This would mean that just disallowing (and detecting) negative feedback is actually sufficient to ensure the declarativeness of selectors in Compose*. Proving that this observation is true for all cases is one of our future works. In terms of implementing an iterative algorithm this observation does not make a big difference: checking for negative feedback already involves trying all the possible orderings of introductions.

Finally, it is important to note that the problems related to negative feedback and circular dependencies can occur in any sufficiently expressive pointcut language, even though it may not be Turing-complete. For example, AspectJ supports introductions based on the *absence* of other elements (i.e. a 'not' operator), as well as dependencies on other introductions. Such a pointcut language is sufficiently expressive to share the problems described in this paper, although implementing a solution may be more straightforward if it is possible to reason about dependencies based on the textual representation of pointcut expressions.

In the next section we explain the algorithm used to resolve dependencies and to apply introductions.

# 6. DEPENDENCY ALGORITHM

This section describes an algorithm that implements the iterative resolution of dependencies, as discussed in the previous section. This means it tries every possible ordering of superimposing the annotations specified in each concern source, while checking for negative feedback and ambiguous end results.

To describe this algorithm, we first need to define a few terms more precisely:

- **(Superimposition) selector**: a selector expression (e.g. $S = \{C|isClass(C)\}$) that returns a set of program elements when evaluated
- **Selector result**: a set of program elements selected by a superimposition selector (i.e. the result of evaluating a selector)
- **(Superimposition) action** (e.g. $S \leftarrow A$): the act of attaching a specific annotation (A) to a set of program elements (selected by S)[3]. An annotation can only be attached once; if a program element already has the annotation A attached, it will not be attached a second time by executing a superimposition action.
- **Iteration**: in every iteration step, exactly 1 superimposition action is executed. All selectors are then reevaluated, rendering new (possibly different) selector results.
- **Negative feedback**: occurs when for any selector result there exists a program element that was selected

[3]In this context superimposition is always about annotations.

in iteration i-1 but not in iteration i. (For i=0 this is never the case, as by definition nothing has been selected before the first iteration.)

- **State**: the current set of selected program elements for each selector, and a list of actions executed to reach this situation.
- **Endstate**: a state where the execution of any superimposition action will not change any of the selector results.

## 6.1 Inputs, outputs, variables

To describe the algorithm, we define its inputs and outputs first.

Inputs are modelled as follows:

```
selectors[0..s]:
  superimposition selectors described by:
  selector name, predicate, result variable

action[0..n]: superimposition actions, described by:
  selector_name, annotation name
```

Also, we define a *State* container object that contains the following information:

```
Set selResults[0..s]:
 for each selector, the set of selected program elements

integer last_action:
 the superimposition action (0..n) that was
 executed to get to this state

integer prev_state:
 a pointer to the state before last_action was executed
```

The output can be either:

- An error condition (exception thrown) when the algorithm detects that negative feedback occurred, or that there are several *endstates* that have different selector results.
- An array of selector results, one for each superimposition selector in the application, representing the final selector results. Also, the algorithm renders a list of annotations and the program elements they should be applied to.

## 6.2 Algorithm description

The algorithm basically implements a breadth-first search: given the initial state (where no actions have been executed yet), it performs all of the possible superimposition actions one by one and adds a new State to a list of states if an action generates selector results that differ from those in the current state *and* the new state does not already occur in the list of states. However, if any of the selector results shrinks by executing an action (i.e. it misses at least one element in the new state that was selected in the current state) the algorithm stops, because this is an error condition (negative feedback between selectors). If executing any action in a particular state does not render different selector results, that state is marked as an *endState*. If there are several end states, it is checked that they all have the same selector results. After it has handled a state, the algorithm tries the next from the list until there are no states left to handle.

Listing 13 describes this algorithm in pseudo-code.

```
dependencyAlgorithm()
{
  number_states = 1; // Total number of states.
  current_state = 0; // Currently handled state.

  // Define initial state
  state[0].selResults = evaluate(selectors);

  while (current_state < number_states)
  {  // Any state left to be handled?
    // assume endstate, until proven otherwise
    currentIsEndState = true;
    for (action = 0..n)
    { // Try every possible action..
      // attach annotations to match current state
      setAnnotationState(state, action);

      newState.selResults = evaluate(selectors);
      if (newState.selResults !=
           state[current_state].selResults)
      { // Selector results changed
        // so this is not an end state
        currentIsEndState = false;
        if (for any i in 0..s:
         newState.selResults[i] misses any elem
         from state[current_state].selResult[i])
          throw NegativeFeedbackException;

        if (for any i in 0..number_states-1:
            newState.selResults !=
                state[i].selResults )
        { // New state, add it to list of states
          newState.last_action = action;
          newState.prev_state = current_state;
          state[number_states++] = newState;
        } // new result found
      } // selector changed
    } // action loop

    if (currentIsEndState)
    { // No action rendered a different result =>
      //  current state is an end state
      if (endstate == undefined or
       endstate.selResults == newState.selResults)
       // Correct end state found
        endState = state[current_state];
      else
        throw DifferentEndResultsException;
    } // found an end state

    current_state++; // handle the next state
  } // state handling loop

  // Set annotations according to the end state
  setAnnotationState(endState);
  // return set of sel. elems. for each selector
  return endState.selResults;
}
```

**Listing 13: Dependency algorithm pseudo-code**

## 6.3 Example run of the algorithm

To demonstrate how the algorithm works in practice, we show an example run for a simple combination of selectors and introductions that can be found throughout the paper.

We take the class *User* as it is introduced in listing 1, and assume the existence of a class *SessionID*. Now, we consider an application that combines the selectors and introductions in listings 7 and 8. In this case, the inputs for the algorithm look as follows (S=Selector, A=Action):

```
S0 :  transientFields = { F |
        typeHasAnnotationWithName(T, 'TransientClass'),
        fieldType(F,T) }
S1 : transientClasses = { AnySess |
       isClassWithName(S, 'SessionID'),
       classInheritsOrSelf(S, AnySess) }
A0 : transientFields <- TransientField
A1 : transientClasses <- TransientClass
```

**Listing 14: Example algorithm inputs**

Figure 1 represents a sample run of the algorithm. The different states that the algorithm encounters are displayed from left to right. Vertically, we can see which program elements are matched by each of the selectors in a given state. The arrows indicate the transitions between states, while the labels next to the arrows indicate what action (introduction) was executed to get from one state to the other.

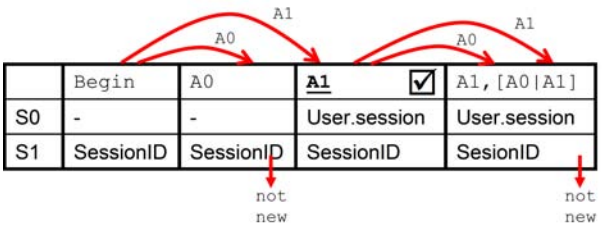


**Figure 1: Example run of the algorithm**

In the initial state, no annotations have been superimposed. The algorithm begins by evaluating all selectors. In this state, S0 (*transientFields*) does not select any fields, as there are no classes that have the annotation *TransientClass* attached. S1 (*transientClasses*) selects the class *SessionID*.

Consequently, the algorithm tries to generate new states by applying the superimposition actions one at a time. In this case, the algorithm starts by executing A0, which superimposes the annotation *TransientField* on fields selected by S0. As S0 does not select anything in the begin state, the resulting state (after applying A0 and reevaluating S0 and S1) is of course identical. If a newly generated state already occurs in the list of states, we do not need to handle it a second time (therefore, it has no outgoing arrows - in practice, it is not even added to the list of states to be handled).

Next, the algorithm tries to superimpose A1, still from the begin state. After it reevaluates S0 and S1, it turns out that S0 now selects the field *User.session* - because executing A1 superimposed the annotation *TransientClass* on the class *SessionID*, and S0 selects fields that have a type with this annotation attached. From the begin state, there are no other possibilities, because we already tried to apply all possible actions (A0 and A1). This state is not an end state, as the application of A1 led to a new state.

The second state is skipped, as it is identical to the begin state. The third state repeats the same process: A0 and A1 are executed again, one at a time. It turns out both actions lead to the same, unchanged, state. Because the algorithm cannot find any action that leads to a new state, this is an end state. The fourth state is skipped, as it is not a new state. At this point the algorithm has reached the end of the list of states, which means it has generated all different introduction orderings and selector results. In this case, we can see that no negative feedback occurred anywhere during the execution of the algorithm (i.e. there is no state where any selector matches less than in its predecessor state). Also, there is only one end state, so there are clearly no ambiguities. Thus, the selector results for S0 and S1 are returned according to the situation found in the end state.

## 6.4 Efficiency of the algorithm

We chose to use a breadth-first search rather than a depth-first (recursive/backtracking) solution because of several reasons:

- The calculation of a new 'state' involves evaluating all selector predicates, which is a (relatively) expensive operation. Hence, we sacrifice memory to gain speed by avoiding duplicate state calculations. We can achieve this by storing states that have been encountered already - a breadth-first algorithm already keeps such a list.
- We have to consider all (different) orderings to check whether the results are the same and to make sure that no negative feedback occurs for any possible ordering. Therefore, it does not matter that backtracking would find a first solution faster. For the same reason, the use of optimizations that would find a first solution faster (e.g. an A*-algorithm) would not make a difference.
- Breadth-first searching can cause problems related to state space explosion (resulting in excessive memory usage). However, we know that most actions will generally return the same selector results, even when using a different ordering. Because we check for duplicate states, most realistic cases are unlikely to cause such a state-space explosion, even if they involve many selectors and introductions. However, our algorithm is essentially a brute-force approach, so it is always possible to construct a worst-case scenario that consumes a lot of time and memory.

## 6.5 Termination of the algorithm

It is not obvious that the algorithm described in the previous section will terminate in all cases. In this section, we show that it does.

Non-termination could be caused by the while-loop in the algorithm. This loop has the exit condition $current\_state < number\_states$ (both values are positive integers). In each cycle, *current_state* is incremented. However, *number_states* can potentially be incremented repeatedly within a single cycle. This could cause the algorithm to never terminate. Therefore, we inspect the circumstances under which *number_states* is incremented. There are 3 possible cases when executing each action within a cycle:

1) An action was executed that made at least one program element disappear from a selector result set (i.e. negative feedback occurred). This is an error condition that will terminate the algorithm.
2) An action was executed that did not change any selector result. This case will be ignored, because it has been handled already. Like in case 1, *number_states* will not be incremented.
3) An action was executed that added at least one program element to at least one selector result. If this results in a case that has not been handled yet (the worst case), *number_states* is incremented.

Only in the third case is *number_states* incremented. In that case we are dealing with a monotonically increasing result set (over several cycles). Also, there is a finite set of program elements that can be in each selector result set (the number of program elements does not grow during the execution of this algorithm). Therefore, case 3 will eventually cease to occur, as there will simply be no program element left to

add to any selector result. This means that eventually case 2 or 1 will occur.

Because *current_state* is increased in every cycle, and eventually no new occurrences of case 3 can be found, the algorithm will always terminate eventually, when *current_state* equals *number_states*.

# 7. RELATED WORK

The benefits of explicitly describing dependencies between annotations are described in [5]. The paper introduces a technique to describe dependencies between annotations, as well as a tool to enforce such dependency relations using a dependency checker tool. The work motivates how the concept of declaring and enforcing dependencies between annotations can be used to model and enforce domain-specific restrictions on top of a common purpose programming language such as C#. Our work focuses on the *derivation* of related annotations, by introducing a technique to not only declare relations between annotations, but also realize the automatic derivation of such relations.

R. Laddad investigates the application of meta-data in combination with AOP in [13]. In this article, he gives practical hints in what situations the application of annotations in combination with AOP (particularly, AspectJ) can be useful. In our paper, we also investigated several new ways of using annotations in combination with AOP, e.g. by allowing the superimposition and derivation of annotations.

The latest versions of AspectJ [1] and JBoss [11] support the use of annotations: join points can be designated based on referring to annotations in the shadow of join points. Similar to the superimposition mechanism in Compose*, the introduction of annotations is also supported in these languages. The main differences are in the expressiveness of the pointcut languages and the way of specifying the introduction of annotations. Both AspectJ and JBoss have only a limited language to select program elements on which annotations can be introduced. In contrast, the selector language of Compose* allows for specifying arbitrary complex queries to select program elements for introductions. Another important difference is that the weaver of AspectJ uses implicitly the order of the declarations of introductions, whereas Compose* does not rely any ordering information by ensuring the declarativeness of the introduction specifications.

The problem of resolving multiple levels of dependencies also occurs in the domain of source code transformations. JTransformer [12] is a transformation tool that uses a language named Conditional Transformations to specify source code transformations. The expression power of this language to specify transformations (i.e. expressing which elements should be transformed) is intentionally limited: it allows for reasoning about dependencies between transformations based on the syntax of the transformation specification (i.e. even without the context of a particular application). This enables the detection of *potential* conflicts between transformation specifications, even if a conflict may not occur in all applications to which such a (potentially conflicting) combination of transformations could be applied. However, the use of a Turing-complete selector language in Compose* did not allow for using a similar approach. Note that there is a trade-off here: the approach of Conditional Tranformations allows for detecting inherent (application independent) conflicts in the specification by offering a transformation language with a limited expression power. On the other hand, Compose* offers an expressive selector language for superimposition, however, our dependency resolution algorithm can detect only application specific conflicts.

A radically different approach to resolving aspect composition problems is taken in [14]. Here, aspects are defined as a declaration of changes to a program (i.e. as transformations). The paper defines a clear approach for determining the order of applying such changes: a global composition specification of all aspects is to be specified, and advices at shared join points are applied in their order of appearance in the aspect definition (source code). Such an approach clearly solves any possible ambiguities and works well in situations where aspects are defined as incremental changes on top of an existing program (and on top of each other). However, to write such a global composition specification a programmer has to know about all the dependencies in an application. This can be troublesome when combining (existing) aspect libraries in a new application. Also, the behavior of an aspect may depend on its location in the composition specification. This reduces the ability for a programmer to understand an aspect as a module on its own, i.e. without knowing what prior transformations may or may not have influenced the pointcuts written in this particular aspect. For this reason, we prefer an approach that keeps pointcuts declarative rather than depending on an imperative ordering specification. When a pointcut specification is ambiguous in combination with other aspects or introductions, we detect this problem.

Finally, the combination of logic languages, negation and (non-)termination has been the subject of much research within the domains of logic programming and databases. There exist examples of logic languages that guarantee declarativeness and termination of predicate resolution, even in the presence of negation. However, such languages always have to impose restrictions on their expressiveness. An example of such a language is DATALOG, which is a subset of prolog; an extensive discussion on its expressiveness, features and extensions to the language can be found in [9].

# 8. DISCUSSION

The superimposition and derivation of annotations as described in this paper has been implemented as a module in Compose*. A limitation in the current version is that parameters of annotations cannot be queried yet. Also, we intend to add support for writing superimposed annotations back to the intermediate language (IL) code (cf. compiled classes, or bytecode, in Java), to support non-aspect oriented frameworks. This functionality has not been implemented yet (i.e. superimposed annotations can only be used within Compose*).

One may consider a case where design information is introduced through superimposition (without derivation rules) and then the occurrences of that same property are used in a pointcut expression to select join points. This is an example style that our approach is not aiming at, as this could have been expressed directly in a single pointcut expression.

Our solution is aimed at keeping the specification of introductions and pointcuts declarative. The alternative is to allow precedence specifications between introductions. Even though this is definitely a solution that can be considered, it has some drawbacks: if pointcut specifications might no longer be declarative in all cases, the programmer needs to look at the context (i.e. other aspects within the same application) in order to understand what will match an expression. Such explicit dependencies might hinder the reusability of aspects. Also, even with such an approach, it would still be desirable to *detect* ambiguities, as allowing for precedence specifications does not ensure that programmers will always know when it is necessary to use them.

# 9. CONCLUSION

The technology for using annotations together with AOP is becoming readily available; more and more aspect-oriented languages support the designation of join points based on references to annotations. The introduction (superimposition) of annotations is not a new idea either; a few aspect-oriented languages already support the introduction of annotations over multiple program elements. However, as we discussed in the section on related work, these AOP languages offer relatively simple static pointcut languages to express the locations where annotations can be introduced.

In this paper, we consider the application of an *expressive* static pointcut language for introductions. This pointcut language is a predicate-based, Turing-complete query language that allows for specifying complex queries as a means to select program elements on which annotations can be introduced. Queries can also select program elements based on the annotations that are already associated to program elements. By introducing annotations through queries that select program elements based on other annotations, we obtain the *automatic derivation* of annotations. By supporting the derivation of annotations, dependent annotations (and complete annotation hierarchies) can be automatically introduced. By ensuring the *declarativeness* of annotation introductions, we believe that we can keep the use of this mechanism as straightforward as possible for the programmers.

However, to ensure declarativeness, it is necessary to handle the dependencies between the evaluation of pointcuts and the introductions of annotations. The main contributions of this paper are related to this issue:

- We analyzed the possible dependencies among introductions and identified cases where dependency problems may arise (section 5).
- Based on this analysis, we developed an approach and designed an algorithm to resolve the above mentioned dependencies, and detect the possible dependency problems (section 6).
- We showed that this algorithm will always terminate either by providing a correct resolution of the dependencies, or detecting ambiguity in the weaving specification (section 6.5).
- As a proof of concept, we have also implemented and tested our approach in Compose*[6], which is our aspect-oriented platform (section 8).

This approach can also be applied in other aspect-oriented languages to the weaving of introductions. For instance, the latest AspectJ weaver implicitly chooses the order in which introductions are applied, in some cases depending on arbitrary criteria such as the ordering of declarations within a source file. Our approach ensures the declarativeness of introduction specifications; this means that by adopting our approach, the weaver would not need to rely on the order of these declarations.

Finally, our approach is also *generic* in the sense that it is applicable to other types of introductions, not only the introduction of annotations. For example, the selection language can be applied to introduce methods in the same way we introduced annotations. When a method is introduced and this method is referred to by an other superimposition specification (i.e. a pointcut), the same dependency issues will arise that we identified at the introduction of annotations.

# 10. REFERENCES


[1] Aspectj - http://aspectj.org/.

[2] AspectJ team. The AspectJ 5 Development Kit Developer's Notebook - http://eclipse.org/aspectj/doc/next/adk15notebook/.

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

[4] L. Bergmans and M. Akşit. Principles and design rationale of composition filters. In Filman et al. [8], pages 63–95.

[5] V. Cepa and M. Mezini. Declaring and Enforcing Dependencies Between .NET Custom Attributes. *In Procedings of the Third International Conference on Generative Programming and Component Engineering (GPCE'04) - LNCS 3286*, pages 283–297, 2004.

[6] Composestar project - http://composestar.sf.net.

[7] *C# language specification*. ECMA International, December 2002.

[8] R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005.

[9] S. Greco, D. Sacca, and C. Zaniolo. DATALOG Queries with Stratified Negation and Choice: from P to DP. In *ICDT '95: Proceedings of the 5th International Conference on Database Theory*, pages 82–96, London, UK, 1995. Springer-Verlag.

[10] K. Gybels. Using a logic language to express cross-cutting through dynamic joinpoints. In *Proceedings of the Second German Workshop on Aspect-Oriented Software Development, Technical Report IAI-TR-2002-1 Universität Bonn*, 2002.

[11] JBoss project - http://jboss.com/.

[12] JTransformer project - http://roots.iai.uni-bonn.de/research/jtransformer/, 2005.

[13] R. Laddad. AOP and metadata: A perfect match - http://www-128.ibm.com/developerworks/java/library/j-aopwork3/. *AOP@Work series*, 2005.

[14] R. Lopez-Herrejon, D. Batory, and C. Lengauer. A disciplined approach to aspect composition. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 68–77, New York, NY, USA, 2006. ACM Press.

[15] I. Nagy, L. Bergmans, W. Havinga, and M. Aksit. Utilizing design information in aspect-oriented programming. In *Proceedings of International Conference Net.ObjectDays, NODe2005*, Lecture Notes in Informatics. Gesellschaft für Informatik (GI), 2005.

[16] Sun Microsystems. Java 1.5 documentation - http://java.sun.com/j2se/1.5.0/docs/.