Introduction and Derivation of Annotations in AOP

Applying Expressive Pointcut Languages to Introductions

Wilke Havinga havingaw@cs.utwente.nl
Istvàn Nagy nagy@cs.utwente.nl
Lodewijk Bergmans bergmans@cs.utwente.nl
University of Twente, Department of Computer Science
P.O. Box 217, 7500 AE Enschede, The Netherlands

ABSTRACT

Meta-data annotations and AOP are a powerful combination. Several aspect-oriented languages already support the use of annotations as a selection criterion in pointcut designators. Some languages also support the introduction of annotations using aspect-oriented techniques. This paper describes a combination of these techniques as they are implemented in Compose*, our aspect-oriented language. The use of a powerful selector language in Compose* is combined with a mechanism to superimpose annotations. The combination of these techniques enables us to specify the derivation of annotations. Derivation relations between annotations lead to dependencies between selector expressions. We investigate in what cases such dependencies are problematic and present an algorithm that resolves the dependencies and detects problematic cases.

1. INTRODUCTION

The concept of annotations has been introduced recently in various programming languages, such as Java[14] and C# [7], to enable the attachment of meta-data to program elements. Although the attachment of annotations does not any have direct influence on the execution of an application, they can be used by compile-time tools or meta-facilities. In this paper, we investigate the use of annotations in aspect-oriented programming, discuss the problems of *superimposing*[1] annotations, and present a solution to these problems in the context of a concrete implementation of the *Compose** language[4][6].

This paper starts with a small example that explains the use of annotations in .NET. In section 3, we show some problems related to the use of annotations in state of the art programming languages. To avoid these problems, we propose to superimpose (introduce) annotations in an aspect-oriented manner. In section 4, we show how the current superimposition language of Compose* can be used to superimpose annotations. However, when annotations can be both superimposed and used as a selection criterion for superimposition certain dependency problems may arise. In section 5, we identify the cases where such dependency problems occur. We introduce an algorithm that resolves the dependencies and detects dependency conflicts when annotations are superimposed. We show that this algorithm terminates in all cases. Finally, we conclude the paper with a related work, discussion and the conclusion.

[1] i.e. *introduction* of annotations in the terminology of AspectJ [2]

2. ANNOTATIONS IN .NET

```
1  [Persistent]
2  class Person {
3    String name;
4    int age;
5
6    [Update] public void setAge(int a){age = a;}
7    [Query] public int getAge(){return age;}
8
9    [Update] public void setName(String n){name = n;}
10   [Query] public String getName(){return name;}
11 }
```

Listing 1: Annotations in .NET (C#)

Listing 1 shows a simple C# class that has annotations attached to a few program elements: the class Person and some of its methods. The notation *[Persistent] class Person* specifies that the program element *Person class* has the *Persistent* annotation attached. Similarly, the *setAge* method has the *Update* annotation attached.

Annotations can be used to express design intentions and semantic properties of program elements. This allows for designating program elements based on their semantic properties, instead of explicitly enumerating a list of elements in pointcut specifications. For example, a persistence concern can easily select only the classes that have the *Persistent* annotation attached, without the need to enumerate (the names of) classes that should be persistent.

In [13], it was pointed out that naming conventions, marker interfaces and other techniques are used in practice to express meta-date in the absence of support for annotations. There it was also show that such techniques reduce the adaptability of applications as well as other desired software engineering properties. However, the use of annotations is not free of problems either, as will be explained in the next section.

3. MOTIVATION

In current (OO) programming language implementations, the annotations are always statically bound to certain program elements, i.e. they are specified directly as part of the source code of a class (as illustrated in the introductory example, listing 1).

This leads to two problems. The first problem is that annotations are scattered over the application source: if every persistent class has the annotation *Persistent* directly bound to its source, the annotations are scattered over the application. In fact, we can state that the annotation *Persistence* is therefore tangled with base classes (such as the class *Person* in this example), because their code cannot be fully separated. On the other hand, it is important to note that in case of certain domains (e.g. security) programmers may intentionally want to bind annotations statically to ensure certain constraints in different applications. In this case, every application that reuses a program unit with annotations will have the same set of annotations.

The second problem is that annotations cannot be application-specific. Suppose the class *Person* 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 *Person*. 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.

The problems of scattering and tangling are at the core of aspect-oriented programming. Our first step toward a solution is, therefore, to reuse an existing (aspect-oriented) solution: by allowing *annotations* to be superimposed at all the places where they should be attached in the source of a concern. Note that similar observations have been made by others, resulting in the ability to superimpose annotations in e.g. AspectJ[1], JBoss[9] and AspectWerkz[3].

A related problem is that annotations may depend on each other. That is, an annotation can be attached to a certain program element, if another—related— program element has a certain annotation. To illustrate this, we refer to the annotations attached to class *Person* in Listing 1. As an example, we could define the following rule for the member variables of a persistent class: if a class has the annotation *Persistent*, then each of its public methods without annotation has to be marked with the annotation *Query* by default. Practically, the attachment of a given annotation may trigger the attachment of every other dependent annotation.

One might ask why it is necessary to attach the annotations *Persistent* or *Create* if their places can be designated by the pointcuts that could express the rules above. Using such pointcuts can be in fact sufficient when these annotations are only used within pointcut expressions of aspects. However, annotations might be used by other tools or frameworks as well. For this reason, we propose to support the derivation of annotations. 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 many cases, this removes the need to manually specify where annotations have to be attached (either in the concern source or the source of the base application).

The next section explains how these observed problems are addressed in our aspect-oriented framework based on .NET, called Compose*.

4. SUPERIMPOSING ANNOTATIONS

This section is divided into three parts: first, we explain how annotations can be used within pointcuts as a criterion for selecting program elements in Compose*. Next, we explain the mechanism of superimposing annotations, and discuss how this helps to solve the problems mentioned in the motivation. Finally, we discuss how join point selection based on annotations can be combined with the superimposition of annotations to provide a technique to *derive* annotations.

4.1 Annotation-Based Join Point Selection

This section explains how the selector language in Compose* supports the use of annotations.

Listing 2 shows part of an example *ObjectPersistence* concern. The selector *persistentClasses* selects all classes that have the *BusinessObj* annotation, using a predicate-based selector language that can select program elements based on the static structure of the application. In this example, all program elements *C* are selected, provided that they are classes that have the annotation *A*, which (as specified on the 2nd line of the selector expression) must be a program element of the annotation type that is named *BusinessObj*.

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

Listing 2: Using annotations in Compose*

This particular selector can be used to implement persistence in a concern separate from the rest of the code, without the need to enumerate all the relevant classes in the concern source.

4.2 Superimposition of Annotations

To solve the problem of scattered, statically bound annotations, we would like to have a mechanism to specify groups of annotations in a separate concern source. In this way, annotations that pertain to a particular concern (such as persistence, synchronization, security) can all be specified in one place: the concern that they belong to. We introduce a simple extension to the existing superimposition mechanism in Compose*: a new language construct that specifies the superimposition of annotations on a set of selected program elements. Note that the selector mechanism itself does not change: program elements can still be selected based on their name, properties and relations to other program elements (i.e. based on the static structure of the application).

As an example, suppose that our application has a generic *DataStore* class which has to be made persistent. This is handled by the *ObjectPersistence* concern, which also marks the object classes that should be persistent—keeping everything that has to do with persistence in one place.

```
1 concern ObjectPersistence {
2   superimposition
3     selectors
```

```
4       dataStoreClasses =
5         { C | isClassWithName(DS, 'DataStore'),
6               classInheritsOrSelf(DS, C) };
7     annotations
8       dataStoreClasses <- Persistent;
9   }
```

Listing 3: Superimposition of annotations

In listing 3, the class *DataStore* and its subclasses are selected by the selector *dataStoreClasses* (line 4-6). The annotation *Persistent* will be attached to this set of selected classes (line 8). If within another application a different set of classes needs to be made persistent, this can be handled easily because the annotation is now decoupled from the base classes that it is attached to. In addition, the annotations are no longer scattered over the application source, as they are specified in one place only: the *ObjectPersistence* concern source. This makes it easier to create more reusable components.

The benefit of this approach is that it is possible to use combinations of static annotations in the (original) application source and superimposed annotations. Thus, concerns can query for annotations that may or may not have been attached by other concerns—the concern itself does not (have to) know in what manner and by whom the annotation was attached. This enables a better separation of concerns, as the dependencies between concerns can now be expressed through annotations. In addition, annotations (even those that are superimposed) can also be read by other tools and frameworks, for example through the use of reflection.

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. To demonstrate what is meant by 'deriving', and show how it can be used, we introduce another example in listing 4, which will also be used in section 5 to explain certain complications.

```
1  concern Synchronization {
2     filtermodule SyncModule {
3       // handle synchronization..
4     }
5     superimposition {
6       selectors
7         updateMethods =
8          { Method | methodHasAnnotationWithName
9                           (Method, 'Update')};
10        queryMethods =
11         { Method | methodHasAnnotationWithName
12                          (Method, 'Query')};
13        syncedClasses =
14         { Class | methodHasAnnotationWithName
15                          (Method, 'Synchronized'),
16                    classHasMethod(Class, Method) };
17
18      annotations
19         updateMethods <- Synchronized;
20         queryMethods  <- Synchronized;
21
22      filtermodules
23        syncedClasses <- SyncModule;
24    }
25 }
```

Listing 4: Derivation of annotations

Listing 4 describes a part of a simple synchronization concern. The goal is to intercept all calls that touch the data inside the object and synchronize calls to these methods[2]. This concern assumes that all methods that read or write the object state have the annotation *Update* or *Query* attached in the original application source. The selector *updateMethods* (line 7-9) selects all methods in the program that have the annotation *Update*, while the selector *queryMethods* (line 10-12) selects all the methods that have the annotation *Query* attached. The *annotations* part (line 18-20) then superimposes the annotation *Synchronized* on the methods selected by either of those selectors. Subsequently, the selector *syncedClasses* (line 13-16) will select all the classes that contain at least one synchronized method, and the filtermodules part of the superimposition specification (line 22-23) superimposes the *SyncModule* on only those classes.

The point of this mechanism is that annotations can be derived based on a combination of other program element properties and relations, including other annotations. This can be useful to generate annotations automatically instead of manually. In this particular case, it is probably still necessary to mark methods as 'update' or 'query' methods by hand (depending on the exact definition of 'update'/'query' and the program elements that can be used within selectors). However, in other cases it is possible to attach annotations automatically, based on the (static) properties of the program itself.

Based on the notion that a method updates object state, other annotations can then be attached as well, as shown in this example.

The derivation of annotations can be seen as a logical consequence of extending the selector language to use annotations, and at the same time, enabling the superimposition of annotations using this extended selector language. However, the derivation of annotations can cause dependencies between selectors, as the result of one selector can depend on the (non-)existence of annotations that may have been attached through another selector. In the next section, we discuss the problems that can be caused by such dependencies.

5. DEPENDENCY PROBLEMS

In the previous section we described a technique to superimpose annotations on program elements. The pointcut expression (selector) that defines these program elements may refer to the presence or absence of a particular annotation. The derivation of annotations may result in dependencies between selectors and superimposition. This section describes the problems that can be caused by such dependencies, and presents a solution approach.

The problem caused by dependencies between selectors is that the order of their evaluation may matter. For instance, in the synchronization example (listing 4), one possible order

[2]Similar to attaching the *synchronized* attribute to a method in Java. However, this way synchronization can be application-specific and could even be implemented using different scheduling techniques, e.g. using different strategies to optimize for updates or queries.

is to first evaluate *updateMethods* and *queryMethods*, then superimpose the annotation *Synchronized* and then evaluate *syncedClasses*. If we apply this order of evaluation, the result is that every update and query method will have the annotation *Synchronized* attached. However, if we evaluate *syncedClasses* before the annotation *Synchronized* is attached, the selector does not match the update and query methods. Hence, the filtermodule *SyncModule* would never be superimposed. As this simple example shows, different orders of evaluation and performing the superimposition can cause different results due to dependencies between selectors.

Note that in this case there is just one level of dependencies, but there could be another concern added, which again depends on the presence of the annotation *Synchronized*. To evaluate the selectors and perform the superimposition statements in the right order, these dependencies have to be detected. However, this means it must be determined which order is the *right* one. One sensible guideline could be: "if selector Y depends on the superimposition of annotation A on program elements selected by selector X, we have to evaluate X and superimpose A on the elements selected by X *before* evaluating Y".

These observations lead to three distinct issues:

1 It must be possible to detect when a selector depends on another selector. However, depending on the expressiveness of the selector language it may not be possible to detect dependencies just by reasoning based on the syntax of a selector.
2 There can exist multiple levels of dependencies, and even circular dependencies might occur. This suggests that some kind of iterative algorithm is needed to resolve the dependencies between selectors. However, if circular dependencies occur, such an algorithm might never terminate.
3 The word *before* implies an ordering. This means that (a) the concern specification is non-declarative, (b) the selector results may be different, depending on the order of evaluation. For these reasons, a correct order of evaluation should be resolved automatically (if possible), otherwise the order should be specified by the user.

It is important to realize that all these issues depend on the expressiveness of the selector language that is used. In the next three subsections we clarify the problems stated above by giving examples for each case, and investigate the root cause of each problem.

5.1 Detecting dependencies

In case the selector language is based on a Turing-complete language, it is impossible to detect dependencies between selectors. Although this allows for powerful reasoning within selectors, it makes it impossible to reason reliably about the results of evaluation by looking at the syntax of selector expressions. This is for example the case in our Compose* compiler[3]. To demonstrate the problem, consider the following selector:

[3]in fact, it is based on Prolog and uses a predefined set of predicates to query a model of the program structure.

```
1 YSharesAnnotWithX =
2  {Y | isClass(Y), hasAnnotation(Y, A),
3       isClassWithName(X, 'X'),hasAnnotation(X, A)};
```

This example selects all classes Y that share at least one annotation (bound to variable A) with class X. However, we do not know *which* annotation is shared by both classes. However, this information is needed to determine whether this selector depends on the superimposition of a specific annotation. It is clearly impossible to infer this from the specification of the predicate. This case is a demonstration of a more general problem: it is impossible to infer information about the execution results (here: what values become bound to each variable) from the syntax of a statement written in a Turing-complete language.

A possible solution would be to restrict the expressiveness of the selector language, such that it is (at least) no longer Turing-complete. However, to make it practically feasible to reason about selector expressions based on their syntax, quite severe restrictions to the selector language are necessary. We do not want to impose such restrictions on the selector language used by Compose*, as we see the powerful reasoning enabled by using a Turing-complete language as a major feature.

Another possible solution is to look at the *results* of selector evaluation; if the result of evaluating a selector changes after we have superimposed a certain annotation, there must obviously exist a dependency. However, the reverse is not true: the fact that the result does not change 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.

5.2 Circular dependencies

The dependency resolution problem can be addressed by iterating over every possible ordering of evaluating the selectors and superimposing annotations, until a fixpoint is reached. That is, the algorithm iterates 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.

However, this leads to the second problem stated above: can we guarantee that such an iterative procedure will terminate? In certain cases, circular dependencies may cause infinite loops. To illustrate such a case, we show an example in listing 5.

```
1 selectors
2   noA = {C|not(classHasAnnotationWithName(C,'A'))};
3 annotations
4   noA <- A;
```

Listing 5: Example of a circular dependency

The selector *noA* selects all classes that do not have the annotation named *A* attached. Next, we specify that annotation A should be superimposed on these classes. After executing the superimposition of this annotation and reevaluating the selector *noA*, it does not match any classes (as all classes that did not have annotation A attached before do have it now). This suggests the superimposition should not actually have been executed. However, if we want to 'undo' the superimposition, *noA* would again match the classes where we removed the annotation. 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; in reality, the selector and superimposition specification may be divided over different concerns. This example demonstrates that we cannot ensure that iteration over selectors and superimposition of annotations will terminate.

If a selector language does not have any means to express negative dependencies, however, annotations would never have to be removed in a later iteration because of the changing selector results (as occurs in the example above). In such a language, the set of results for each selector would eventually become 'saturated', as the number of both program elements and annotations that can potentially be superimposed is finite. Thus, the algorithm would always reach a fixpoint.

Based on these observations our conclusion is that infinite loops can occur if and only if the selector language supports any kind of 'not' or 'exclusion' operator. Unfortunately, removing all types of exclusion operators from our selector language is not an option: the ability to specify negative dependencies is sometimes a very useful feature—a good example would be to specify that classes that do *not* have any *Transient* fields automatically get the annotation *Persistent* superimposed.

5.3 Ambiguous selector specifications

The existence of exclusion operators in the selector language leads to the third and last problem: superimposition specifications and selectors are preferably 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. We demonstrate this by the following example:

```
1 selectors
2   noA= {C|not(classHasAnnotationWithName(C, 'A'))};
3   noB= {C|not(classHasAnnotationWithName(C, 'B'))};
4 annotations
5   noA <- B;
6   noB <- A;
```

Suppose this concern is added to a program that has a class with neither annotation A nor B attached. Thus, this class is selected by both *noA* and *noB*. If we superimpose the annotation *B* on it first (line 5) and then reevaluate the selectors, the class does not match selector *noB* anymore, so annotation A will never be attached. However, if we first superimpose the annotation A (line 6), it does not match selector *noA* anymore, so annotation B will not be attached. In this case, the end results are different (either annotation A or annotation B is attached). The specification is clearly ambiguous, as there is no way to discern which order of evaluation was intended by the programmer.

5.4 Summary

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

- The expressiveness of the selector language in Compose* does not allow reasoning about dependencies based on the syntax 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. A similar approach (in an AOP context) was taken in [11].
- 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. 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 are the foundation 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 5 would be detected as problematic). Note that this does not limit the expressiveness of the selector language itself: it is still possible to use 'not' and other types of exclusion constructs. Additionally, we do not allow for different results based on different orders of attaching the annotations. In this way, ambiguous specifications (such as in listing 5.3) will also be considered an error case. To detect such cases, our algorithm tries every possible ordering.

Adding these restrictions does not only solve the problems that have been mentioned in this section, but also makes sense from a conceptual point of view: it is much easier (for the programmer) to understand what is actually selected by each selector, yielding a more robust language. In the next section we explain the algorithm.

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. Meanwhile it checks 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 *selector result*.
- *Selector result*: a set of program elements selected by a superimposition selector (i.e. the result of evaluating a selector).
- *(Superimposition) action* : the act of attaching a specific annotation (A) to a set of program elements (selected by S)[4]. An annotation can only be attached once; if a program element already has an 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 in iteration i-1 but not in iteration i.
- *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
```

In addition, 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.

[4]In this context we only consider superimposition of annotations; we believe though, that the algorithm is equally suitable for other program elements such as methods, fields and filter modules

6.2 Algorithm description

The algorithm basically implements a breadth-first search: given the beginstate (where no actions have been executed yet), it performs all of the possible superimposition actions 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 previous 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 endstates, it is checked that they all have the same selector results. After it has handled a state, the algorithm tries the next state from the list until there are no states left to handle.

```
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].selResults[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 endstate found
        endState = state[current_state];
      else
        throw DifferentEndResultsException;
    } // found an endstate

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

  // return set of sel. elems. for each selector
```

```
55   return endState.selResults;
56 }
```

Listing 6: Dependency algorithm pseudo-code

6.3 Termination of the algorithm

In this section, we show that this algorithm will terminate in all cases. Non-termination could be caused only by the while-loop in the algorithm (all for-loops have fixed number of finite iterations, there is no recursion 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 an 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). 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 be nothing 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 explained 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 addresses 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 [12]. 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 showed some 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 [9] support the use of annotations: join points can be designated by referring to annotations corresponding to those 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 pointcut language (i.e. type patterns with the support of inheritance) for selecting program elements into which annotations can be introduced. In contrast, the selector language of Compose* allows for specifying arbitrary complex queries to select program elements for introductions. This selector language also provides the ability to specify dependencies between annotations, hence enabling the automatic derivation of annotations.

The problem of resolving multiple levels of dependencies also occurs in the domain of source code transformations. JTransformer [10] 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 Transformations 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.

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 IL code (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 a programming style that our approach is not aiming at, as this could have been expressed directly in a single pointcut expression.

The use of derivation of annotations in several concerns could make it hard for a programmer to keep track of what will match a certain selector expression. However, by ensuring the declarativeness of selector expressions and superimposition of annotations, we believe that we could keep the use of this mechanism as straightforward as possible for the programmers.

Finally, we observed that by disallowing negative feedback, the order of superimposing the annotation can never lead to different end results. We consider proving this observation as one of our future work. This would mean that just disallowing (and detecting) negative feedback is actually sufficient to ensure the declarativeness of selectors in Compose*—the check for different end results may be superfluous.

9. CONCLUSION

Nowadays, the technology for using annotations together with AOP is becoming available, as more and more aspect-oriented languages support the designation of join points through annotations. The introduction (superimposition) of annotations is not a new idea either; a few aspect-oriented languages have already started to support the introduction of annotations over multiple program elements. However, as we discussed in the related work section, these AOP languages offer relatively simple static pointcut languages to express the locations where annotations (or other program elements, such as fields or methods) can be introduced. To support the introduction of annotations with an expressive pointcut language in Compose*, first we analyzed the use of annotations and identified problems related to their usage in .NET. Based on this analysis, we proposed a mechanism in Compose* to superimpose (i.e. introduce) annotations with an expressive static pointcut language. We achieved this pointcut language by adopting the current selector language of Compose*, which is a predicate-based, Turing-complete query language. This language allows for specifying complex queries to select program elements on which annotations can be superimposed. Queries can also select program elements based on the annotations that are already attached to program elements. By superimposing annotations through queries that select program elements based on other annotations, we could achieve the automatic derivations of dependent annotations. However, the consequence of this derivation technique is that there will be dependencies between the evaluation of queries and the superimposition of annotations. We analyzed these dependencies and identified cases where dependency problems may arise. Based on this analysis, we designed an approach and an algorithm to resolve the above mentioned dependencies and detect the possible dependency problems. We showed that this algorithm will always terminate either by providing a correct resolution of the dependencies, or detecting if the superimposition specification is ambiguous.

Note that the proposed superimposition mechanism is generic, since the selection language is not only applicable to the introduction of annotations but also to other type of introductions. For example, the selection language can be applied to introduce methods in the same way as we introduced annotations. When a method is introduced and this method is referred to by another superimposition specification, the same dependency issues will arise that we identified at the introduction of annotations. For this reason, the proposed algorithm is also applicable to resolve these dependencies or detect the possible problems in the superimposition specification.

By extending Compose* to enable the superimposition of annotations, it is possible to separate the annotations from implementation classes, thus preventing the scattering of annotations. Also, this enables better separation between concerns, as they can select program elements based on the existence of annotations that may or may not have been attached by other concerns. By supporting the derivation of annotations, dependent annotations (and complete annotation hierarchies) can be automatically superimposed. In this way, the inconsistencies caused by the manual attachment of such annotations can be avoided as well.

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] AspectWerkz project - http://aspectwerkz.codehaus.org/.

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

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

[11] G. Kniesel, P. Costanza, and M. Austermann. JMangler—A powerful back-end for aspect-oriented programming. In Filman et al. [8], pages 311–342.

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

[13] I. Nagy, L. Bergmans, W. Havinga, and M. Aksit. Utilizing design information in aspect-oriented programming. In *Proceedings of International Conference NetObjectDays, NODe2005*, Lecture Notes in Computer Science, Erfurt, Germany, 2005. Springer-Verlag. to be published.

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