Generic, Property Based Queries for Evolvable Weaving Specifications

Istvan Nagy, Lodewijk Bergmans, Gurcan Gulesir, Pascal Durr, Mehmet Aksit

TRESE group, Dept. of Computer Science, University of Twente
P.O. Box 217, 7500 AE, Enschede, The Netherlands
+31-53-489{5682, 4271}

{nagyist, bergmans, g.gulesir, durr, aksit}@ewi.utwente.nl

ABSTRACT

In the current aspect-oriented languages, advices and pointcuts are explicitly associated in general. This results in weaving specifications that are less evolvable and need more maintenance during the development of a system. To address this issue, we propose *associative access* to advices and aspects: a designating mechanism that allows for referring aspect/advices through their (syntactic and semantic) properties in advice-pointcut bindings. First, this paper presents an extensive analysis of the advice-pointcut binding mechanisms of the state-of-the-art AOP approaches. Based on this analysis, we extend the current weaving (superimposition) specification of our aspect-oriented approach, Compose*. In the new specification, we apply queries that can designate filtermodules and other type of units (e.g. annotations) based on their properties. As an evaluation of our work, we present a tradeoff analysis about the new weaving specification with respect to several software engineering properties, in particular expressiveness, evolvability and comprehensibility. Finally, the paper ends with related works and conclusion.

1. INTRODUCTION

Advices and pointcuts are one of the most important elements of aspect-oriented languages [8]. Advices associated with pointcuts form a large part of the weaving specification by describing the places and subjects of weaving. We argue in this paper that the way these elements are associated has significant influence on the weaving specification with respect to software engineering properties, such as expressiveness, evolvability and comprehensibility [15].

The current advice-pointcut mechanisms of AOP languages maintain explicit dependencies to advices and aspects. This results in weaving specifications that are less evolvable and need more maintenance during the development of a system. We believe that this issue can be addressed by providing *associative access* to advices and aspects instead of using explicit dependencies in the weaving specification. To this aim, we propose to use a designating (query) language in advice-pointcut bindings that allows for referring aspect/advices through their (syntactic and semantic) properties.

This paper is structured as follows: section 2 presents an extensive analysis of the advice-pointcut binding mechanism of various AOP approaches. Section 3 discusses our aspect-oriented approach, Compose* [6][4]. In the first subsection we give details about the current binding mechanism of Compose*. In the following subsections, we show our proposal to extend the superimposition specification of Compose* based on the analysis we performed before. These subsections also contain tradeoff analysis in which we evaluate the proposed mechanisms in view of the above mentioned software engineering properties. Finally, we close the paper with related work and contribution in section 4.

2. An Analysis of Weaving Specifications

In the following sections, we look at the weaving specifications of the state-of-the-art AOP approaches, such as AspectJ [1], AspectWerkz [3] and JBoss [10]. In particular, we examine how certain elements (e.g. pointcuts, advices, etc.) of the weaving specification are associated with each other, how these elements can be reused in another weaving specification, what can be subjects of a weaving specification and to what degree can the weaving specification follow the evolution of concerns. By reflecting on these issues, our goal is to identify those language features that may have significant impact on a weaving specification.

2.1 Associating Advices with Pointcuts

Considering the coupling between the advices and pointcuts, we identified two main categories of the advice-pointcut bindings:

Strong Coupling

The definition of an advice already contains the pointcut to which the advice is permanently bound.

The advice construct of AspectJ is a typical example for this type of association. Figure 1 shows a simple example of binding a before advice to a pointcut called tracedMethods.

```
pointcut tracedMethods(): execution…;
before(): tracedMethods(){
   if (TRACELEVEL == 0) return;
   if (TRACELEVEL == 2) callDepth++;
   printEntering(thisJoinPoint.getSignature());
   …
}
```

Figure 1. Advice-Pointcut binding in AspectJ

Strong association has a negative impact on the reusability of advices and aspects. The main problem is that an advice is permanently bound to a pointcut. This has two consequences: (1) the advice cannot be associated with a new pointcut (i.e. with new joinpoints); (2) the code of an advice cannot be reused from other advices, since an advice cannot be called like an ordinary method; it is only executed when the join point is reached by the control flow.

To circumvent the first problem, AspectJ has the notion of *abstract pointcut* that allows for deferring the specification of a pointcut that is already associated with an advice.

```
abstract aspect Tracing{
   abstract pointcut tracedMethods();

   before(): tracedMethods(){
      if (TRACELEVEL == 0) return;
      if (TRACELEVEL == 2) callDepth++;
      printEntering(thisJoinPoint.getSignature());
      …
   }
   …
}
aspect GUITracing extends Tracing{
   pointcut tracedMethods():
      within (com.app.gui.*) && …;
}
```

Figure 2. An example for an abstract pointcut in AspectJ

In the example of Figure 2, the before advice is associated with the abstract tracedMethods pointcut in the abstract Tracing aspect. This abstract pointcut is concretized in the GUITracing aspect that inherits from Tracing. By applying this technique, the before advice can be associated with various pointcuts when it is necessary. The only disadvantage of this approach is that the comprehensibility of the code decreases, especially when the size of the project scales up. This reason is that every subaspect should be read to determine the complete set of pointcuts (and joinpoints) that the advice is associated to.

To avoid the second problem and provide code that can be reused by several advices, we need to move the code of an advice into ordinary methods that can be called from any advice. We illustrate this in Figure 3 that shows the previous example after the corresponding refactoring.

```
abstract aspect Tracing{
   abstract pointcut tracedMethods();
   before(): tracedMethods(){
      Tracer.traceEntry(
           thisJoinPoint.getSignature());
   }
   …
}
public class Tracer{
  public static void traceEntry(String sign){
      if (TRACELEVEL == 0) return;
      if (TRACELEVEL == 2) callDepth++;
      printEntering(sign);
      …
   }
}
```

Figure 3. Using helper methods in AspectJ

In this example, the code of the before advice is replaced by a call to the traceEntry method of the Tracer class that contains the original code of the advice. As a result, the method can be called (i.e. reused) from different advices. Note that the information from the thisJoinPoint variable has to be extracted in the advice and passed as a parameter to the method, since this variable is known only in the context of an advice. Another important issue is that an around advice cannot be refactored in this way; the reason is similar: the *proceed* keyword is applicable only within the context of around advices.

Practically, abstract pointcuts and helper methods can be considered as useful techniques when one wants to create reusable advices; however, they also had some disadvantages in certain cases, as we have seen in the examples.

Loose Coupling

Advices and pointcuts are weakly associated if they are specified independently from each other and coupled in a new binding specification.

The advice binding construct of AspectWerkz is a typical example for weak association. Figure 4 shows the implementation of the tracing example in AspectWerkz.

```
public class Tracer{
  public void traceEntry(JoinPoint thisJP){
      if (TRACELEVEL == 0) return;
      if (TRACELEVEL == 2) callDepth++;
      String signature =
         thisJP.getSignature().getName();
      printEntering(signature);
      …
  }
}
<aspectwerkz>
   …
   <pointcut name="tracedMethods"
             expression="…" />
   <aspect name="TracerAspect" class="Tracer">
      <advice name="traceEntry" type="before"
              bind-to="tracedMethods"/>
   </aspect>
</aspectwerkz>
```

Figure 4. Advice-pointcut binding in AspectWerkz

In AspectWerkz, aspects are represented by classes and the methods of a class representing an aspect act as advices. In the example above, the traceEntry method of the Tracer class contains the crosscutting functionality that is usually implemented by an advice. The xml descriptor contains a pointcut definition called tracedMethods and an aspect mapping. The aspect mapping declares that the Traces class will be treated (e.g. instantiated) as an aspect in the system, while the advice binding connects the traceEntry method as a before advice to the previously defined pointcut. Note that this aspect mapping allows for reusing a class (that acts as an aspect) in the realization of different aspects (e.g. by using different instantiation strategies). Similarly, using this type of advice-pointcut association allows for binding a method to different (types of) advices. As a result, it is possible to reuse not only pointcuts but also the crosscutting functionality, represented by the method, for the implementation of different advices.

2.2 Multiplicity in Bindings

Considering the multiplicity of advices in the pointcut-advice binding specifications, we have identified two types of bindings:

Many-to-One Binding

In the current AOP languages, the general case is that one pointcut specification (referring to many joinpoints) is associated with one advice in the binding specification.

The advice construct of AspectJ can be considered as a good example for the many to one binding. In AspectJ, advices are always defined with a pointcut reference (or with the pointcut definition itself). For example, Figure 1 shows a definition of a before advice bound to pointcut called tracedMethods. Note that a pointcut can be bound to an arbitrary number of advices; however, a new binding specification should be defined for each binding. Figure 5 shows an example for this: two before advices, located in different aspects, are bound to the same pointcut in two advice definitions.

```
aspect TracerAspect{
   pointcut tracedMethods(): execution…;

   before(): tracedMethods(){
      if (TRACELEVEL == 0) return;
      if (TRACELEVEL == 2) callDepth++;
      printEntering(thisJoinPoint.getSignature());
      …
   }
}
aspect AnotherTracerAspect{
   before(): TracerAspect.tracedMethods(){
      /* another advice bound to the same pc */
      …
   }
}
```

Figure 5. Two advices bound to the same pointcut in AspectJ

The same problem appears in AspectWerkz, as well; a pointcut can be bound to several advices but we need to create a new binding specification for each case.

```
<pointcut name="tracedMethods"
          expression="…"/>

<aspect class="TracerAspect">
   <advice name="traceEntry" type="before"
           bind-to="tracedMethods"/>
</aspect>

<aspect class="AnotherTracerAspect">
   <advice name="traceEntry" type="before"
           bind-to="tracedMethods"/>
</aspect>
```

Figure 6. Two aspects bound to the same pointcut in AspectWerkz

Figure 6 shows an example for this. First, a pointcut called tracedMethods is defined; this is followed by two aspect mappings that contain two advice bindings. Note that in the whole binding specification (aspect mapping + advice binding) we refer to not only advices but also the aspects that contain the advices.

There are certain cases when it is necessary to weave not only one aspect/advice but a set of aspects/advices at a given join point. Clearly, the many to one type of binding specification is not sufficient in these cases. Whenever a new aspect/advice should be bound to a given pointcut, a new complete binding specification should be created for it as well. As a result of this, as Figure 6 shows, we will get a set of binding specification that contains (partially) *duplicated* information that makes the code difficult to maintain.

Note that share join points may appear when we bind a set of advices to the same pointcut, since advices will be woven to the same join point. We may also need to provide control over the execution order of advices to avoid possible conflicts.

Many-to-Many Binding

In the advice-pointcut binding specification, a pointcut can be bound to a set of advices. Although this sounds as an obvious alternative, not too many aspect-oriented languages support this idea.

JBoss provides a mechanism by which many-to-many bindings can be created between a pointcut and several advices and at the same time, controls the execution order of advices, as well. Using the previously introduced example, we illustrate this mechanism in Figure 7.

```
public class TracerAspect{
   Object traceEntry(Invocation object)
                                  throws Throwable
   {…}
}

public class AnotherTracerAspect{
   Object traceEntry(Invocation object)
                                  throws Throwable
   {…}
}

<stack name="Tracing">
  <advice name="traceEntry"
          aspect="TracerAspect"/>
  <advice name=" traceEntry"
          aspect="AnotherTracerAspect"/>
</stack>

<pointcut name="tracedMethods" expr=… />

<bind pointcut="tracedMethods">
       <stack-ref name="Tracing"/>
</bind>
```

Figure 7. Enumerating advices in the binding spec. in JBoss

TracerAspect and AnotherTracerAspect are implemented by classes, as previously. Both classes have the methods (traceEntry*)* that will behave as advices when these classes are mapped to aspects. Subsequently, the corresponding aspects are listed with the corresponding advices between the stack tags. The order of the listed aspects and advices does matter, since they are executed in the order of listing when the join point they are attached to is reached. This whole stack (called Tracing) is bound to the tracedMethods pointcut in the binding specification, defined within the bind xml tags.

Note that this technique is only an *enumeration* similarly to the binding of AspectWerkz, since the aspects are listed within the *stack* tags. On the other hand, using the stack structure for enumerating the aspects has some benefits as well. First, the stack acts a 'virtual' module for a set of aspects that are wrapped into it. Thus, a set of aspects can be referred through only one reference in the binding specification. (In other words, a set of aspects can be bound to several pointcuts through only one reference.) Secondly, as aspects are organized into a stack structure, the execution order of their advices is also provided for shared join points.

However, there is a problem with using enumeration (or similar techniques, like the stack) in many-to-many bindings: the binding specification should be modified whenever a new aspect is involved in the binding specification (or an existing one is removed). In other words, the enumeration based techniques does not provide *evolvable* binding specifications.

2.3 Associative Access

In general, aspects or advices can share common properties. For instance, aspects with similar functionality can have a common (semantic) property that denotes their analogues semantics. By taking the well-known example of aspect-oriented programming, we can classify aspects with monitoring functionality as *monitoring* aspects. Similarly, other semantic properties can be easily associated with aspects or advices, considering various issues, such as their domain (e.g. *security, persistence* aspects), or even their implementation (e.g. *singleton* aspects). As another

well-know example, we can mention the classification used in the AspectJ Programming Guide [17]: *development, production* aspects.

By involving these (semantic) properties in the binding specification, aspects can be *designated for weaving based on their properties*. For instance, we assume a simple example: we would like to bind the development aspects to a given pointcut. That is, we would like to bind (& weave) every development (monitoring, security, etc.) aspect to a certain join point. By referring to aspect/advices through their semantic properties, we can provide *associative access* to aspects (or advices). This has two positive consequences. First, the expression power of the weaving specification increases. Secondly, if we designate aspects based on their properties, the weaving specification becomes more evolvable and less fragile to changes. Currently, none of the existing aspect-oriented languages provides designating mechanisms that can be used to designate aspects for weaving. In section 3, we show how Compose* can be extended to designate aspects for superimposition[1].

Note that in case of designation we may also get automatically shared join points, like we had it when we enumerated or partially duplicated the binding specifications. As we mentioned earlier, this issue should be addressed in order to avoid possible conflicts.

2.4 Subjects of Weaving

In the current AOP approaches, it is rare that the weaving specification is limited to bindings only between advices and pointcuts. In fact, it is becoming more and more general that other type of 'actions' than 'advices' can be assigned to pointcuts, as well. Introductions or introducing annotations are typical examples for such actions. Similarly to advices, these 'actions' will be crosscutting when they have to be executed over multiple places[2].

In Figure 8, we show some illustrative example from AspectJ and JBoss where the subjects of pointcut binding are not advices.

```
/*--- AspectJ Examples ---*/

pointcut register():
   call(void Registry.register(FigureElement));
pointcut canRegister():
   withincode(static * FigureElement.make*(..));

declare error:
   register() && !canRegister(): "Illegal call";

declare annotation:
   org.xyz.model..* : @BusinessDomain;

/*--- JBoss Examples ---*/

<annotation-introduction
   expr="contructor(Foo->new(..))">
     @org.jboss.single("hello")
</annotation-introduction>

<introduction expr="class(org.acme.*)">
   <interfaces>java.io.Serializable</interfaces>
</introduction>
```

Figure 8. Different type of actions bound to pointcuts in Aspect and JBoss

The first example of AspectJ is the declare error statement. This construct has the compiler signaled an error with the specified message ("Illegal Call") if the join point specified by the pointcuts (register() && !canRegister()) occurs. The declare annotation construct, in the newest release of AspectJ [18], specifies that all types defined in a package with the prefix org.xyz.model have the @BusinessDomain annotation. Similarly, the first JBoss example introduces the @org.jboss.single annotations with the specified argument into every constructor of the Foo class. The second example forces every class in the org.acme package to implement the java.io.Serializable interface.

2.5 Summary

As a result of the analysis, we have identified the following features that can improve the weaving specification:

Loose coupling provides reusable crosscutting behavior: advices and aspects can be reused in different applications, since they can be specified independently, and assigned to pointcuts later. For the same reason, loose coupling is also suitable for developing aspect libraries.

Many-to-many bindings yield more expressive binding specifications: as opposed to many-to-one bindings, a single specification can express the binding between a set of advices and a pointcut. (That is, we can avoid creating a new specification for each binding between an advice and the pointcut.)

Associative access allows for designating a set of advices and aspects, based on their properties (or relationships to other units). Hence, the weaving specification is more evolvable and less fragile to changes, since advices and aspects are implicitly bound to pointcuts, through their properties.

Weaving subject polymorphism: in section 2.4 we observed that several types of language elements can be bound to pointcuts. This can lead to increased expressiveness of the language, as well as improved uniformity.

Our goal is to provide associative access and support various subjects such as filtermodules and annotations in the weaving specification of Compose*. To realize these features, we will present an extension to the current superimposition specification of Compose*.

3. Extending Compose*

The following section gives details about the current weaving (superimposition) specification of our aspect-oriented approach, Compose* [4][6]. In section 3.2 and 3.3 we show our proposal to extend the superimposition specification of Compose*. These sections also contain a tradeoff analysis in which we evaluate the proposed mechanisms.

3.1 The Superimposition Specification of Compose*

Figure 10 presents a simple example case for the superimposition specification of Compose*.

```
concern Tracing{
```

[1] The composition of aspects with the base classes is called superimposition in the terminology of Compose*.

[2] Typically, these places are statically determinable joinpoints.

```
   filtermodule SimpleTracing{ … }

   implementation in Java; …
}
concern Profiling{
   filtermodule SetsCounting{ … }

   implementation in Java; …
}
concern WeaveDevelopmentAspects{
   …
   superimposition{
(1) selectors
       figureClasses =
        { FClasses |
          isClassWithName(Class,'FigureElement'),
          inInheritanceTree(Class,FClasses)
        };
(2) filtermodules
      figureClasses <- SimpleTracing,SetsCounting;
   }
   …
}
```

Figure 10. Enumerating filtermodules for superimposition in Compose*

Using the AspectJ terminology, we apply two *development* concerns in this example: Tracing and Profiling. Tracing contains a filtermodule[3] (SimpleTracing) that implements the crosscutting functionality, i.e. tracing the execution of methods, in this case. Similarly, Profiling also has a filtermodule specification (SetsCounting) to realize the profiling feature: counting the changes of member variables through update methods (e.g. methods with *set* prefixes). Since we are focusing on the aspect/advice-pointcut composition in this paper, we omitted to show the implementation of these filtermodules. The third, independent concern called WeaveDevelopmentAspects contains the weaving specification (using the terminology of Compose*, it is called superimposition specification). The superimposition specification consists of two parts: a selector definition and a filtermodule binding. In Compose*, every selector has a name (figureClasses), which is unique within the concern it is defined. This selector designates (=) a set that consists of possible values for a variable (FClasses), where the predicates after the '|' puts constraints on these values. The first predicate (isClassWithName) binds the class FigureElement to the Class variable. The second predicate binds every class inherited from FigureElement to the FClasses variable by using unification. In the filtermodules part of the superimposition, the SimpleTracing and SetsCounting filtermodules are superimposed on each class that is designated by the previously defined selector (i.e. every class inherited from FigureElement). Thus, every instance of these classes will have an instance of the filtermodules superimposed.

According to the classifications we presented in section 2, the filtermodule binding specification of Compose* can be described as *many-to-many* binding with *loose coupling*. However, we can also see in the example of Figure 10 that filtermodules are only enumerated in the binding specification. As we discussed, the enumeration technique results in difficult code maintenance and not evolvable advice-pointcut bindings. For this reason, in the following sections, we show a proposal to extend Compose* with a new binding mechanism that can provide evolvable weaving specifications.

3.2 Querying Filtermodules

Our goal is to provide associative access to filtermodules, instead of enumerating them. To achieve this, we apply selectors, the existing query mechanism of Compose*[4], to designate filtermodules based on their properties. This is illustrated in Figure 11 by a simple example.

```
concern Tracing{
   filtermodule SimpleTracing{ … }
   filtermodule AdvancedTracing{ … }

   implementation in Java;
   …
}

concern WeaveTracingModules{
   …
   superimposition{
    selectors
       figureClasses =
        { FClasses |
          isClassWithName(Class,'FigureElement'),
          inInheritanceTree(Class,FClasses)
        };
       tracingModules =
          { FModule |
            isFilterModule(FModule),
            inConcern(Concern,FModule),
            isConcernWithName(Concern,'Tracing')
          };

    filtermodules
      figureClasses <- tracingModules;
   }
   …
}
```

Figure 11. Quering filtermodules in Compose*

The example starts with a concern specification, Tracing, that has two filtermodules: SimpleTracing and AdvancedTracing. The second concern, WeaveTracingModules, contains the superimposition specification. The first selector (figureClasses) is already known from the previous example; it will query all the classes inherited from the FigureElement class. The second selector (tracingModules) will query all the filtermodules within the *Tracing* concern. In the filtermodule binding, we bind the tracingModules selector to the figureClasses selector. This means that every instance of the selected classes will have an instance of all filtermodules queried by tracingModules. In this way, selectors are used with two purposes: (a) designating the classes on which the filtermodules are being superimposed; (b) designating the filtermodules that we will superimpose.

Tradeoff Analysis

+ Instead of enumerating the filtermodules one by one, we were able to designate a set of filtermodules based on their properties (or relationships to other units). Hence, the weaving specification becomes *more expressive* with the application of queries both on the base classes and the superimposed filtermodules.

[3] A Compose* unit that represents a set of advices in terms of AspectJ

[4] Selectors use a generic, predicate based language by which we can formulate queries to designate various program elements, such as concerns, classes, methods, fields, etc. involved in a Compose* project.

+ When a new filtermodule is introduced within the *Tracing* concern, it is automatically captured by the query; therefore, the weaving specification becomes *more evolvable* as well.

- If we designate and bind a set of filtermodules to a selector then we will get automatically shared join points. To avoid this problem we have to handle this issue in parallel by providing a superimposition order for filtermodules. We have addressed this issue in a previous work [15].

- The programmer of filtermodules (or aspects) should be aware of the selectors that can potentially capture a newly introduced filtermodule. It might be intended to query the new filtermodule but it might not. Note that the enumeration of the filtermodules represented explicit dependencies to the filtermodules in the weaving specification. By changing enumerations to queries we turned these explicit dependencies into implicit ones, since queries refer to filtermodules through their properties. For this reason, we say that the weaving specification becomes *less transparent* (for programmers) or *less comprehensible*. Tools and inteligent IDEs (*integrated development environments*) can help to resolve these implicit dependencies and show them explicitly. For instance, the AJDT project [1] is a typical example for such a tool.

- So far, we have used only syntactical properties (e.g. a *name* of a concern) and structural relationships (e.g. a filtermodule *contained by* a concern) to formulate queries in the examples. However, if we have to modify (e.g. refactor) the code for certain reason the queries that have worked until now may fail to work in the future. For instance, by renaming the Tracing concern, the tracingModules selector will not work in the last example. That is, the *fragile pointcut* problem [13] may apply not only to the pointcuts that designate joinpoints in the base code but also to the 'pointcuts' (or queries) that designate the aspect/advices for weaving. In the following section, we propose to use annotations on filtermodules to handle this problem.

3.3 Annotating Filtermodules & Concerns

Annotations (aka. *custom attributes* in .NET [5], or *metadata facility* in Java [11], can be generally used to associate (semantic) properties with a language unit. In [14], we showed how pointcuts can exploit annotations applied in the base code. We also presented a limited mechanism to query filtermodules based on annotations attached to filtermodules. In this section, we show how filtermodules can be designated by using the generic query mechanism of Compose*. Figure 12 presents an illustrative example.

```
[Development]
concern Tracing{
   filtermodule SimpleTracing{ … }

   implementation in Java;
   …
}

[Development]
concern Profiling{
   filtermodule SetsCounting{ … }

   implementation in Java;
   …
}

concern WeaveDevelopmentAspects{
   …
   superimposition{
    selectors
       figureClasses =
        { FClasses |
          isClassWithName(Class,'FigureElement'),
          inInheritanceTree(Class,FClasses)
        };
       developmentModules =
        { FModule |
           isFilterModule(FModule),
           inConcern(Concern,FModule),
           hasAnnotationWithName(Concern,
           'Development')
        };

    filtermodules
      figureClasses <- developmentModules;
   }
   …
}
```

Figure 12. Querying filtermodules for superimpospostisition based on their annotations

In Figure 12, we use the same example that we used in Figure 10, in section 3.1. Tracing and Profiling are both development aspects, so we attach the Development annotation to them. In WeaveDevelopmentAspects, we introduce a new selector, developmentModules, which queries all filtermodules contained by a concern with the Development annotation. In the filtermodule binding specification, we bind this selector to the previously defined figureClasses selector. As a result, all filtermodule of each Development aspect will be superimposed on the selected classes.

Tradeoff Analysis

+ Whenever a new concern is introduced with the *Development* annotation, its filtermodules are automatically involved in the weaving specification. In addition, the concerns can have now arbitrary names, since the query does not refer to their names but to their semantic property, realized by an annotation (Development) in this case[5]. The usage of semantic properties makes aspects, especially pointcut expressions and queries, less vulnerable to changes of the program. They provide a larger degree of evolvabilty, since they allow for avoiding dependencies of pointcut expressions upon the structure or naming conventions of the program.

+ The dependencies between program elements can now be made more precise and *easier to understand* by referring to semantic information (design intentions) instead of structure or naming (if the latter would exactly express the intention, it would be preferred, though).

- Disciplined programming is required to keep the correct semantic properties associated with the appropriate program elements. For example, the programmers should not forget to attach the required annotation.

- Similarly, it is important that software engineers have to use a consistent and coherent set of semantic properties for each sub domain of an application (whether from a technical/solution domain, or from the application/problem domain). For instance, if

[5] Tagging program elements manually by annotations is only one possibility to attach semantic properties. Other more advanced techniques are discussed in [14].

programmers use terms such as 'setter', 'writer', or 'updater' inconsistently, our approach has limited value.

3.4 Introducing Annotations

As we discussed in section 2.4, not only advices can be bound to pointcuts. Similarly, not only filtermodules can be bound to selectors (i.e. queries) in Compose*. In this section, we illustrate how queries can be used for introducing annotations. Figure 13 presents a simple weaving specification that introduces an annotation into a set of classes and interfaces.

```
concern AssignBusinessDomain{
  superimposition{
    selectors
      allModelTypes =
       { ModelType |
         isNamespaceWithName(NS,'org.acme.model'),
         contains(NS, ModelType)
       };
    attributes
       allModelTypes <- [BusinessDomain];
  }
}
```

Figure 13. Introducing a single annotation in Compose*

The allModelTypes selector will query all types (both classes and interfaces) in the specified namespace. In the annotation binding part of the specification (after the attributes keyword[6]), we bind the BusinessDomain annotation to every type selected by allModelTypes.

Queries can also be utilized in case of introducing multiple attributes that have no arguments. The approach is similar to the binding of multiple filtermodules. Figure 14 illustrates this by an example.

```
namespace org.acme.sec.annotations{
  [Documented]
  public class Authorized: Attribute {}

  [Documented]
  public class Authenticated: Attribute {}

  public class Encrypted: Attribute {}

}
concern AssignSecurity{
  superimposition{
    selectors

      resources =
       { ResourceClass |
         isClassWithName(RootClass,'Resource'),
         inInheritanceTree(ResourcClass,RootClass)

       };
      securityAnnotations =
       { SecAnns |
         isNamespaceWithName(NS,
               'org.acme.sec.annotations'),
         isClassWithName(Attr,'Attribute'),
         contains(NS, SecAnns),
         inherits(SecAnns, Attr),
         hasAnnotationsWithName(SecAnns,
               'Documented')
       };

    attributes
        resources <- securityAnnotations;
  }
}
```

Figure 14. Querying and introducing multiple annotations in Compose*

In the org.acme.security.annotations namespace there are three annotations defined. The first two are tagged with the Documented meta-annotation. The superimposition specification consists of two selector definition and an annotation binding. The resources selector will query all classes inherited from Resource. The securityAnnotations selector will query every annotation within the org.acme.sec.annotations namespace that has the Documented annotation. As a result of binding securityAnnoations to resources, the Authenticated and Authorized annotations will be introduced in every class inherited from Resources.

Note that this approach cannot be applied when the annotation has arguments, e.g. Author("John Smith"). For such a case, the previous style of annotation introduction (or the enumeration technique, in case of multiple annotations) can still be used.

4. Conclusion

4.1 Related Work

In section 2, we have discussed the weaving specification of AspectJ [2], AspectWerkz [3] and JBoss [10] in details. Both AspectJ and JBoss provide facilities for weaving other subjects (e.g. annotations and introductions) than advices. The advantage of our approach over the weaving specification of these languages is the ability of selecting advices and aspects that are bound to pointcuts. Currently, none of above mentioned languages offers such a mechanism.

In the most recent version of AspectJ [18], it is possible to attach annotations to aspects and advices. Thus, a pointcut can designate the execution of an advice based on an annotation. However, unlike our approach, advices still cannot be selected and bound to pointcuts based on this information.

JQuery [9] is a flexible, query-based source code browser, developed as an Eclipse plug-in. In JQuery, users can define their own queries to select certain elements of a program. The JQuery query language is defined as a set of TyRuBa [7] predicates which operate on facts generated from Eclipse JDT's abstract syntax tree. The predicates of JQuery are dedicated for Java and the factbase of JQuery is based on Java sources and bytecode files. In Compose* we also use a predicate language to formulate queries for defining selectors. Our predicates are dedicated for Compose* and the factbase is based on the repository model of a Compose* project.

4.2 Contribution

In this paper, we presented an extensive analysis of the pointcut-advice binding mechanisms of various aspect-oriented language. Based on this analysis, we proposed an extension to the current superimposition specification of Compose*. In the new specification we applied queries, founded on a generic, predicate-based language, that allow for designating both the places (join

[6] We use the keyword attributes instead of annotations, since annotations are called custom attributes in the terminology of .NET.

points) where we want to weave, and the units (e.g. filtermodules, annotations) that we want to weave[7].

As a result, the weaving specification is more expressive and evolvable compared to the analyzed binding mechanisms. By applying annotations[8] (semantic properties), we improve the evolvability of the weaving specification to a greater extend; additionally, the intention of the composition is easier to understand. On the other hand, the weaving specification becomes less transparent (comprehensible) for programmers. Also, we observed that the use of annotations requires disciplined programming to bind the correct semantic properties to the appropriate program element. We believe that both of these issues can be addressed by tool support. Intelligent IDEs can improve the transparency problem by resolving and showing the implicit dependencies among the units involved in the weaving process. The "human error factor" related to use of annotations can be lessened by using reasoning mechanisms and tools that can *automatically* derive annotation specifications, for example, based on control-flow or data-flow analysis. We consider addressing these issues in our future work.

5. Acknowledgement

We would like to thank Wilke Havinga and Joost Noppen for their useful comments and discussing various parts of this paper.

6. REFERENCES

[1] AJDT project: http://eclipse.org/ajdt/
[2] AspectJ project: http://aspectj.org
[3] AspectWerkz project; http://aspectwerkz.codehaus.org
[4] Bergmans, L, & Aksit, M., Principles and Design Rationale of Composition Filters, in: R. Filman, T. Elrad, S. Clarke, M. Aksit (eds.), *Aspect-Oriented Software Development*, Addison-Wesley, 2004
[5] C# Language Specification, in *Standard ECMA-334 2nd Edition*, ECMA International, December 2002.
[6] Compose* project: http://composestar.sf.net
[7] De Volder, K., Type-Oriented Logic Meta Programming, *Ph.D Dissertation*, Vrije Universiteit Brussel, Programming Technology Lab, 1998.
[8] T. Elrad, M. Aksit, G. Kiczales, K. Lieberherr, H. Ossher, "Discussing Aspects of AOP", *Communications of the ACM*, Volume 44, Issue 10, October 2001.
[9] Janzen, D., de Volder, K., Navigating and querying code without getting lost, in *Proceedings of the 2nd international conference on Aspect-oriented software development*, pp. 178-187, Boston, Massachusetts, 2003
[10] JBOSS project: http://www.jboss.org
[11] JSR 175 Public Draft Specification: A Metadata Facility for the Java Programming Language, Sun Microsystems, Inc., 2002-2003.
[12] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten M., Palm, J., Griswold G. W., An Overview of AspectJ, in *Proceeding of the 15th European Conference on Object-Oriented Programming*, LNCS, Springer-Verlag, London, United Kingdom, 2001.
[13] C. Koppen and M. Stoerzer. PCDiff: Attacking the fragile pointcut problem. *1st European Interactive Workshop on Aspects in Software*, Sep 2004.
[14] Nagy, I., Bergmans, L., Towards Semantic Composition in Aspect-Oriented Programming, *1st European Interactive Workshop on Aspects in Software*, Berlin, Germany, September 2004
[15] Nagy, I., Bergmans, L., and Aksit, M., Declarative Aspect Composition, *2nd Workshop on Software-engineering Properties of Languages for Aspect Technology,* Lancaster, UK, March 2004.
[16] Newkirk, J., Vorontsov, A.A. How .NET's Custom Attributes Affect Design, *IEEE Software*, September/October 2002.
[17] The AspectJ Team, *The AspectJ™ Programming Guide*, 1998-2001 Xerox Corporation, 2002-2003 Palo Alto Research Center.
[18] The AspectJ Team, *The AspectJ™ 5 Development Kit Developers's Notebook*, http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/aspectj-home/doc/ajdk15notebook/index.html, December 10, 2004

[7] Different types of units require different implementations of weaving.

[8] In [14], we have already discussed the application of annotation in AOP and their benefits.