

# Towards an open weaving process

Bart De Win, Bart Vanhaute and Bart De Decker  
Departement of Computer Science, K.U.Leuven  
Celestijnenlaan 200A, B-3001 Heverlee, Belgium  
bartd, bartvh, bart@cs.kuleuven.ac.be

August 3, 2001

## Abstract

In order to find the right balance between flexibility and expressiveness, current aspect-oriented technologies provide a predefined set of selection and combination mechanisms to drive the weaving process. If a programmer wants to use specialized mechanisms beyond the possibilities of this set, he must resort to a combination of mapping onto existing mechanisms and hand coding into the application. Establishing a fixed set of mechanisms considerably limits the usefulness of aspect-oriented programming both at weave-time and at run-time. As a result, building advanced reusable systems becomes hard.

In this position paper we propose a first step towards a more open weaving process for AspectJ by introducing a generic way to describe joinpoints through a two step process of enumeration and restriction. By means of some examples, we show that this technique enables very flexible deployment decisions. And although the ideas are not fully mature yet, we also try to discuss the advantages and disadvantages of the approach.

**Paper category: A or F**

## 1 Introduction

In our work, we are trying to design a framework of aspects for adding application level security mechanisms to a program. In [2, 3] we presented the current results of our work and we briefly discussed some limitations of the state-of-the-art AOP technology in that scope. This position paper provides a more in depth discussion of the latter. *Comment for the reader: due to time limitations we were not able to polish the paper as much as we would have liked to. In particular, the section on generic pointcut description and the discussion might be improved a bit more. We hope that the paper is clear enough to understand the general idea.*

In the aspect-oriented paradigm, a program consists of different parts, corresponding with different viewpoints of a problem. These parts are at some point in time (e.g. before compile-time, at compile-time or at load-time) composed into one artefact based on their relationship. Aspect relations are typically described in a special purpose composition language operating over the static or

dynamic structure of the program. This language provides a predefined set of primitives that can be combined using logical operators.

Unfortunately, building advanced reusable systems sometimes requires specialized composition features, not covered by the predefined set of primitives. For example, it would be hard or even impossible to describe that all subclasses of a particular class that override a specific inherited method would have to be secured more than others. Moreover, composition does not only depend on the program structure, but it might require knowledge about other, external information, both at weave-time and at run-time. The desired encryption algorithm might for instance depend on the host where the program is compiled. Even worse, it might depend on the user that is authenticated at run-time. Currently, this kind of information must be hand coded into aspects although it is actually part of the composition logic.

Without sacrificing the expressiveness of aspect-oriented programming, we would like to have more composition flexibility. We think that an open weaving process, where one can intervene in the composition and actually program his own composition rules, would be highly beneficial. However, it is always important to keep in mind the balance between flexibility and expressiveness and we do not want to end up with an ultra flexible MOP that is hard to program in practice.

In the rest of this paper, we first present a new view on the current composition model of AspectJ. Based on this model, we then propose a more generic composition model, which enables an open weaving process. To illustrate this idea, we provide some examples that clearly demonstrate the increased flexibility of the model. We end the paper by discussing the advantages and disadvantages of our approach.

## 2 Composition model

In AspectJ, dynamic composition<sup>1</sup> of aspects is based on the idea of joinpoint sets, specified through the definition of pointcut descriptors. AspectJ offers a predefined set of keywords, or primitives, to define such pointcut descriptors. Looking at these primitives in more detail shows that they can actually be decomposed into two basic operations:

1. An **enumeration** produces a set of joinpoints or references to specific places inside a unit of composition, being the program structure or, more recently, the execution of the program. Depending on the granularity and the strength of the language, different types of joinpoints can be supported, such as method executions, object creations or one could imagine even single lines of code. In AspectJ, *calls(\*)* and *receptions(\*)* are examples of such enumerations.
2. A **restriction** imposes a certain constraint on the elements of a sets, which means that a joinpoint must satisfy the constraint in order to be part of the set. As such, it is used to limit the number of joinpoints in a set. Restrictions by themselves make no sense. They must be applied

---

<sup>1</sup>Dynamic composition as opposed to static composition where extra functionality is actually copied into existing classes through the definition of introductions.

to sets, which means that they should always be linked to enumerations. Examples of restrictions in AspectJ are *instanceof(type)* and *within(type)*. Furthermore, the negation of each enumeration can also be considered as a restriction and vice versa. Moreover, strictly speaking, a primitive such as *calls()* with a concrete parameter (not a wildcard) should be regarded as a restriction on the enumeration.

Each of the primitives in AspectJ can be mapped onto (specific implementations of) these orthogonal basic operations and it would actually be a good exercise to do so.

The combination of basic primitives in AspectJ can also be revisited in the scope of these two basic operations. For enumerations, they can be considered as basic mathematical operations on sets. The `&&` and `||` operators in AspectJ correspond respectively with intersection and union. Note that the intersection of sets is only useful if there exist joinpoints with the same type in both sets<sup>2</sup>, while the union of sets can be applied at all times. For restrictions, `&&` and `||` are used to mandatory or optionally combine the restrictions. As a sidenote, the `&&` operator is also used in AspectJ to attach restrictions to enumerations. Since this inter-operation functionality is clearly different from its intra-operation counterpart, it might be a good idea to replace the notation of this operator by something else in order to make this distinction more clear, as proposed further in this paper.

This basic operation model, in which the current composition model of AspectJ seems to fit, is the cornerstone of a more generic and flexible composition model that we will present in the next section.

### 3 Generic composition and open weaving

For a more generic composition model, we propose to extend the AspectJ language by adding two new keywords, one for each of the basic operations described in the previous section.

#### 3.1 Enumeration

As defined earlier, an enumeration produces a set of joinpoints inside a unit of composition. A generic form of enumeration must uncouple this ability from the description of how to find them. For example, one can imagine the use of predefined keywords for this purpose, but it should also be possible to search them one by one based on a specific algorithm. To enable this, a representation of the structure of the program must be available to the body of the enumeration. This could for instance be provided by means of an abstract syntax tree of the program.<sup>3</sup>

For generic enumerations, we propose the following notation:

---

<sup>2</sup>In AspectJ, it would also make no sense to define the following pointcut: `receptions(foo(*)) && calls(bar(*))`.

<sup>3</sup>We think that it is sufficient to limit this representation to weave-time, because a run-time representation would be more difficult to reason about. Moreover, the functionality of the latter is already offered by existing MOP's.

<set of points> **eval** {<expression>}

where expression is the body of the enumeration that specifies how to find the joinpoints. The result of the evaluation of the body is contained in a set of joinpoints. Using this keyword, creating a set containing all executions and receptions could look like this:

```
eval{
  for each i in AST.methods() do begin
    add execution(i) ;
    add reception(i);
  end ;
}
```

Note that the power of this evaluation mechanism depends on the mechanisms provided to browse through the abstract syntax tree and on the supported types of joinpoints.

## 3.2 Restriction

A restriction imposes a certain constraint on a set of joinpoints, which means that a joinpoint must satisfy the constraint in order to be part of the set. Similar to a generic enumeration, generic restrictions should allow to program the specific constraint criteria. As input, it requires the considered joinpoint. The output is true or false depending on whether it satisfies the restriction.<sup>4</sup> For generic restrictions, we propose the following notation:

true|false **if** {<expression>}

An example of the restriction to only allow methods with a specific name, could look like this:

```
if{return (this.type==method and this.name=="SpecificName") ;}
```

This generic restriction enables composition based on other factors, such as information from the environment. As an example, it could depend on the host where the program is weaved, as illustrated in this restriction:

```
if{return getHostName().equals("MyComputer") ;}
```

Even more powerfull, the composition could use on run-time information. For this purpose, disposal of the current aspect instance would be highly beneficial, as illustrated in the next restriction, where the composition relies on the user that is authenticated at run-time:

```
if{return this.username.equals("UntrustedUser")}
```

In this example, *this* points to the current aspect instance where the pointcut was defined.

---

<sup>4</sup>Note that another point of view might provide a set of joinpoints as input and return the same set possibly reduced. However, programming a restriction becomes harder this way and it is less appropriate for run-time restrictions, since at run-time the set of restrictions might not be available anymore.

Run-time composition raises the question when the restriction must be evaluated. For some restrictions, this can be performed at weave-time, for others, only run-time evaluation will provide enough flexibility. The latter actually requires pointcuts to be represented at run-time<sup>5</sup>. This is clearly a trade-off between efficiency and flexibility: evaluating all restrictions at run-time is the most flexible solution, but it will not be very efficient and vice versa. We recommend to evaluate the restrictions as soon as possible. Maybe the programmer could help the weaver with this decision by means of a special keyword.

### 3.3 Open weaving

For combinations of enumerations and restrictions, the language must allow to attach restrictions to certain enumerations. We think it is a good idea to introduce a new symbol, %, for this functionality. A combination of different enumerations and restrictions would then look like this:

```
pointcut p: ((eval{} && eval{}) % if{}) && (eval{} % if{});
```

To evaluate such expressions, enumerations should be evaluated and combined first. Afterwards, restrictions should be applied to the elements of the sets they restrict. As discussed before, this might be performed at run-time.

Given this two generic keywords, we see an *open weaving process* as one where only the keywords *eval* and *if* are defined. The existing keywords in AspectJ can be decomposed into a combination of enumerations and restrictions and can as such be considered pure syntactic sugar. Of course, an aspect tool could provide a library of keyword implementations to simplify the task of the programmer.

As a remark, we think that the current pointcut description keywords in AspectJ are very practical and usefull. Dividing them into pure enumerations and several restrictions might complicate things considerably. Therefore, we should consider allowing to program restriction code into enumerations, as the following example in pseudocode illustrates:

```
eval{
  p = class("ParticularParent");
  for each c in classes(){
    if p.subclass(c) and c.contains("SpecificMethod")
      then add execution(c.method("SpecificMethod"));
  }
};
```

## 4 Discussion

The advantages of deploying a more generic composition model are considerable. For the users of aspect-oriented technologies, the added value of the approach is the increasing flexibility. As demonstrated by the examples in the previous section, this enables advanced composition rules that can depend on the program structure and on other information available at weave-time and at run-time.

---

<sup>5</sup>This also raises the question if pointcuts could be defined static. We will not elaborate on this matter here.

Furthermore, this new composition model is an important step towards runtime customization of aspects, as described in [1]. For the designers of aspect-oriented tools, this model provides a stable and permanent base to implement their system. Besides new joinpoint types, the introduction of new keywords will no longer require changing the tool, but it will suffice to add an extra keyword implementation to the keyword repository.

On the downside, an increase in flexibility always requires a more complex model. Since the current pointcut description model of AspectJ is very practical, it is hard to justify this profound change of working. However, we feel that the gain of flexibility warrants this change. Moreover, the current keywords can coexist with enumerations and restrictions, as long as the former are translated to the generic model of the latter at weave-time. Furthermore, much of the complexity of pointcut programming could be anticipated by providing a library with a set of powerful keyword implementations.

In this work, there still remain some open issues. First, it is difficult to predict when (weave-time or run-time) restrictions should be applied. Restrictions at run-time necessitate the existence of pointcut instances. As mentioned before, this also raises the question whether the introduction of static pointcut descriptors could be useful. Second, should we only allow strict enumerations or would it be more useful to allow coding restrictions inside enumerations. If the latter is more appropriate, how do we then define a model without redundancy? Finally, although we have a good feeling about it, we actually don't know whether this idea will turn out to be implementable or not. Feedback of the different tool providers and of all other interested people would help us considerably.

## References

- [1] F. Matthijs, W. Joosen, B. Vanhaute, B. Robben, and P. Verbaeten. Aspects should not die. Position paper at the ECOOP '97 workshop on Aspect-Oriented Programming.
- [2] B. Vanhaute, B. De Win, and B. De Decker. Building Frameworks in AspectJ. Position paper at the ECOOP'01 workshop on Advanced Separation of Concerns.
- [3] B. De Win, B. Vanhaute, and B. De Decker. Security through Aspect-Oriented Programming. To appear in the proceedings of the first working conference on Network Security, November 2001.