A Meta-Level Specification and Profile for AspectJ in UML

Joerg Evermann
Victoria University Wellington
Wellington, New Zealand
jevermann@mcs.vuw.ac.nz

ABSTRACT

Aspect-oriented programming (AOP) has become a mature technology. Increasingly, calls for support of AOP on the software model level are being voiced. This paper addresses these calls by offering a meta-model of AspectJ in UML. Using the UML extension mechanisms, this meta-model is at the same time a profile to support AspectJ modelling in UML. In contrast to previous work, this profile offers complete meta-level integration of all AspectJ concepts. The use of standard XMI based modelling allows the use of the profile in commercially available CASE tools and supports easy code generation.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specification—*Languages*; D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.3.3 [**Programming Languages**]: Language Constructs and Features

1. INTRODUCTION

While aspect-oriented programming (AOP) is rapidly maturing, there is not yet any de-facto or de-jure standard for aspect-oriented software modelling. The de-facto software design language, UML [14], does not offer specific constructs for aspects and their associated concepts. However, UML does provide standardized extension mechanisms that can be used to provide aspect modelling facilities. In UML 2.0 these extension mechanisms have the power to make any meta-level model a profile.

In this paper, we present a meta-model of the AspectJ language extensions. Using the powerful extension mechanisms, this meta-level model *is* a UML profile. This approach offers some advantages over existing proposals.

First, it is supported by UML 2.0 compliant modelling tools. The extension requires no special software tool, and allows aspect modelling to be used within mature software tools. For example, the work described in this paper was developed using the commercially available tool MagicDraw, version 11.5. This is in contrast to earlier proposals, which are not based on profiles and extend UML either by introducing new meta-model classes, or new notation elements, or both. Those proposals cannot be used with available modelling tools and require specific tool support.

Second, because the proposed technique is supported by UML XMI model interchange facilities, the model extension, as well as any models it has been applied to, can be exchanged between different MOF compliant UML tools.

Third, the proposal allows all aspect-related concepts to be specified in meta-model terms, no textual specification is necessary. This means that the models can be easily manipulated or verified, without having to parse textual specifications.

Fourth, the proposal maintains strict separation of base-model and crosscutting concerns in the models it is applied to, the primary motivation behind AOP.

This proposal is not an aspect modelling extension for generic AOP. The conceptual differences between different aspect implementations, e.g. Aspect# and AspectJ are substantial. We have focused on providing a UML extension to support AspectJ because of the maturity of the development of AspectJ.

2. RELATED WORK

An overview of some of the prior work for modelling aspects in UML is presented in [16]. The early work presented at AOM 2002 is based on the extension mechanisms in UML 1.x versions. Because these mechanisms are not fully integrated with the meta-model, the specification of advices and pointcuts often remains in textual form. The connection between aspects and base-model is made as part of the model, instead of an aspect extension to the model, thereby foregoing the clear separation of base-model and crosscutting concerns that is central to AOP.

Initial work presented in [1] proposed the specification of aspects as stereotypes on classes and was later extended [2] to include advice and pointcut specification. It models crosscutting associations to show which aspect features relate to which base-model elements. The proposal in [15] is not defined in meta-model terms and uses special keywords in a textual specification of roles to define pointcuts. It is limited to advice on method calls and field accesses. An aspect stereotype for UML collaborations was developed in [11], however without being fully defined in UML meta-model terms. An earlier profile for AspectJ [17] represents advices and pointcuts as stereotyped operations, and the connection to the base features is made via dependencies in the model. Similarly, the proposal in [4] uses textual specification of pointcuts, rather than being based on (meta-)model elements. Later extensions to this in [6] are similar to our proposal in that aspects are stereotyped classes. Again, because no meta-model based profile is developed, the connection between aspects and base-model is made as part of the model, rather than an aspect extension to it. An inital proposal for aspect modelling using UML 2.0 was presented in [3], however without fully defining an extension profile, as we do here.

Other prior work is based on defining new UML meta-classes, instead of defining stereotypes for existing meta-classes, which requires specialized tool support for the new meta-classes. Instead of using the `extends` relationship type in UML, these proposals use the `generalization` relationship type to define the new aspect concepts. The research in [8] introduces new UML meta-classes and therefore requires specialized tools for their support. A meta-model for generic AOP is offered in [5], but with no apparent mapping to AspectJ and without describing an implementation. A full meta-model based approach, similar to this proposal, is found in [18]. However, rather than employing the standard lightweight extensions of UML, this approach also introduces new meta-classes, thus requiring specialized tools.

Other work on aspect modelling in UML proposes join point annotations for UML [9]. A translation of aspect UML to object-oriented Petri-nets is described in [12] but is limited to pointcuts around method calls. Weaving on the model level is presented in [10] as part of work on design-by-contract. Code generation from aspect-extended UML models is presented in [7] who opt against the XMI based method proposed in this paper and instead use custom tool extensions.

3. ASPECT UML EXTENSION

The main point of distinction of this work to previous proposals is the focus on developing a complete and comprehensive meta-model of AspectJ. It also resolutely employs a meta-model based specification. For example, the operations selected by a call join point are specified as operation elements of the model, not as a textual specification, as previous work has done.

This allows the integration of aspect features with base-model features on the meta-model level, rather than as part of the model, as previous work was forced to do, and thereby maintains strict separation of base and crosscutting concerns. For example, the application of an aspect to a classifier is not shown by any kind of relationship in the model. Instead, analogous to the specification in AspectJ, pointcuts of an aspect select specific operation elements or attribute elements of the model (Sec. 4 and Figs. 5, 6).

This section presents a meta-model of the AspectJ concepts. It is modelled on the UML meta-level, so that it is usable as a profile (Fig. 1). Rather than specializing UML meta-classes, as previous work has done, we extend the existing meta-classes. A UML stereotype is a meta-class which enters into `extends` relationships with existing meta-classes [14]. Visually, this is shown with the extended class in square brackets (Fig. 1). Attributes that are modelled on stereotype meta-classes will translate to tags when the profile is applied [14]. Similarly, values of stereotype attributes will become values of tags when the profile is applied [14]. This extension mechanism in UML 2.0 is therefore a very powerful way in which any meta-level model immediately becomes usable as a profile. The following paragraphs present the UML meta-model for each AspectJ construct.

CrossCuttingConcern. While not directly specifiable in AspectJ, we introduce the meta-class `CrossCuttingConcern` as a way of grouping related aspects. `CrossCuttingConcern` extends the UML meta-class `Package`, because a cross cutting concern contains aspects in the same way as packages contain classes. Because the UML meta-model already specifies that packages own classes, the `CrossCuttingConcern` meta-class does not need to be associated with the `Aspect` meta-class.

Aspect. An aspect contains both static features (that do not specify behaviour), such as pointcuts, and dynamic features (that specify behaviour), such as advices. Furthermore, aspects can be specialized, and can realize interfaces. These characteristics are sufficiently close to the features of a UML class, so that we model aspects using a meta-class `Aspect`, which extends the meta-class `Class`. This makes the `Aspect` meta-class a stereotype on the UML `Class` construct.

Aspects have some properties that are not already offered by classes. These are modelled as attributes of the meta-class, which will become tags when the profile is applied to a model. A boolean attribute `isPrivileged` indicates whether the aspect is privileged. A multi-valued attribute `declaredParents` allows the declaration of generalizations by the aspect ("declare parents: I extends J" in AspectJ). Because the extension is on the meta-level, the data type for this attribute is the meta-class `Generalization` in UML. In other words, the values of this sterotype-tag pick out generalization model elements when this extension is applied. Similarly, `declaredImplements` allows the declaration of interface realizations ("declare parents: I implements J" in AspectJ). The data type of this attribute is the UML meta-class `InterfaceRealization`.

Because aspects can be instantiated per pointcut, the attributes `perType` and `perPointCut` are used to specify the type of aspect instantiation and associated pointcut. The data type of `perPointCut` is the `PointCut` meta-class (i.e. the stereotype) in this extension and the values for `perType` are provided by the enumeration `AspectInstantiationType`.

Aspect precedence is modelled as a recursive relationship between aspects. Because the precedence ordering in AspectJ is total, each aspect has at most one directly preceding and following aspect.

Advice. With `Aspect` being a meta-class that extends `Class`, the dynamic features of aspects, i.e. advices, play the role of class behavior: The meta-class `Advice` is an extension on the meta-class `BehavioralFeature`. Behavioural features include collaborations and state charts, implying that these can also be used to specify advice implementations, as proposed in [1, 2]. Because behavioral features are owned by classes in the UML meta-model, `Aspect` does not have to be associated with `Advice`. We add the constraint that the «`Advice`» stereotype can only be applied to behavioral features of classes that are stereotyped «`Aspect`».

The UML meta-model already associates operations with signatures. Hence, our extension does not need to model

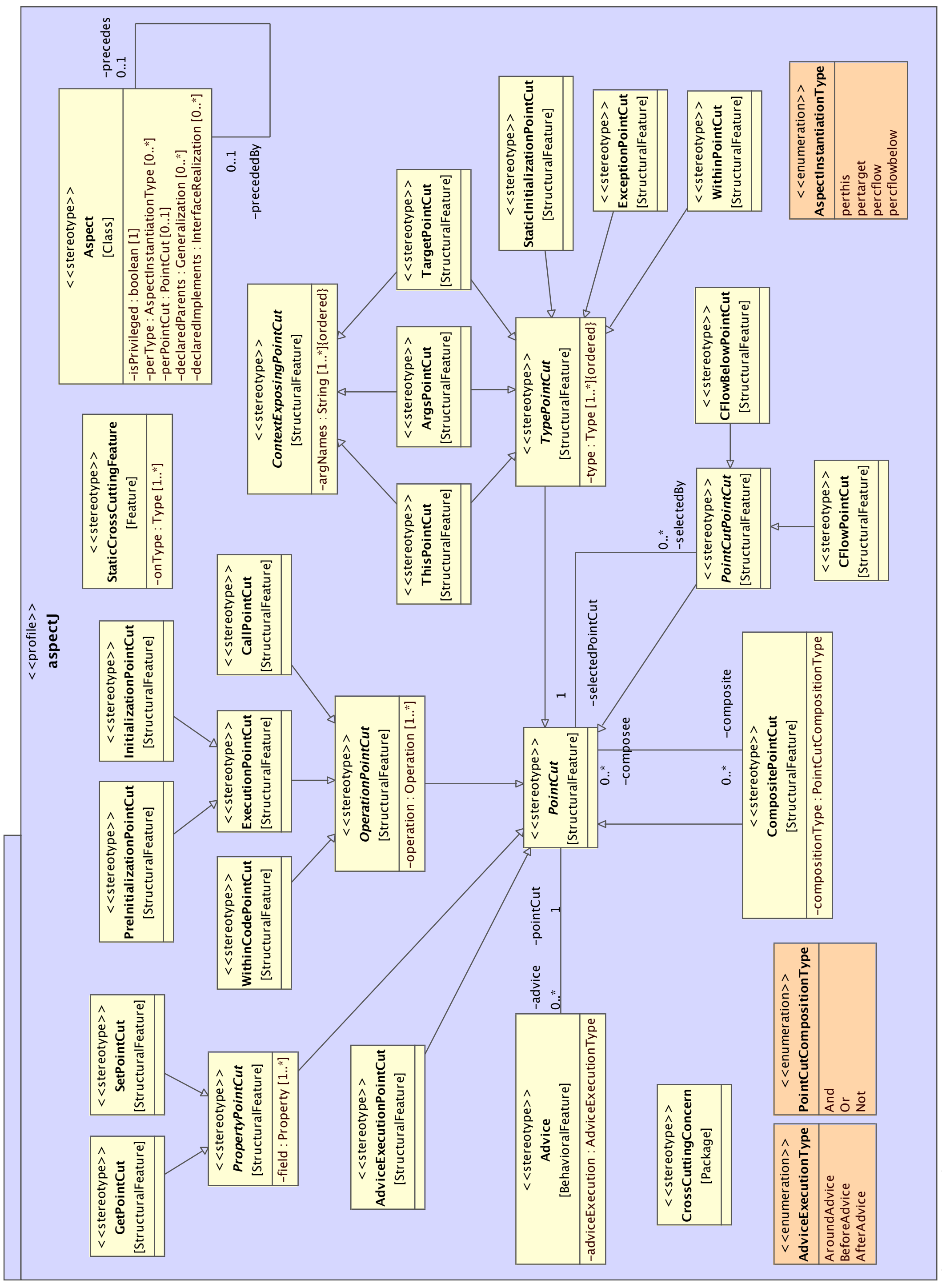


Figure 1: AspectJ profile for UML

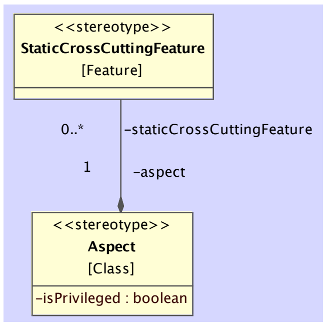


Figure 2: Alternate model for Static Crosscutting Features

signatures for advices. When applying the «Advice» stereotype to an operation, we get parameters, return values, and raised exceptions automatically.

Advice code can be executed before, after, or instead of (around) a pointcut. We model `adviceExecution` as an attribute of the `Advice` meta-class. The values are provided by the enumeration `AdviceExecutionType`. Because each advice has a signature by virtue of being an operation, the signature implies whether an ”after” advice is ”after returning” or ”after throwing” by examining the operation’s return parameter and raised exception.

Static Crosscutting Features. Aspects may introduce new features to existing classes and types. Because such crosscutting features can be static or dynamic, the meta-class `StaticCrossCuttingFeature` extends the UML meta-class `Feature`, which is the superclass of both `Property` and `Operation`. The cross cutting features are owned by the aspect (by virtue of the ownership of attributes and operations by classes), there is no need to associate `StaticCrossCuttingFeature` with `Aspect`. We add the constraint that the «`StaticCrossCuttingFeature`» stereotype can only be applied to features of classes that are stereotyped «`Aspect`».

To specify on which types the crosscutting feature is to be introduced, the `StaticCrossCuttingFeature` meta-class possesses a multi-valued attribute `onType` whose data type is the UML meta-class `Type`.

We have chosen to model crosscutting features as owned by the aspect, rather than by the classifier they are introduced on. While this requires the extra meta-model element `onType`, it enforces the separation of base-model and crosscutting concerns that is fundamental to AOP. The alternative would be to associate the meta-class `Aspect` with the meta-class `Feature` so that the aspect can pick out any feature owned by any classifier in the model (Fig. 2). The application of this alternate meta-model in Fig. 3 shows that in this case the static crosscutting features are visually modelled as part of the base-model element rather than the aspect, thereby giving up the clear separation of concerns into the aspects.

PointCut. A pointcut does not specify dynamic behaviour. Hence, the meta-class `PointCut` extends the UML meta-class `StructuralFeature`. We add the constraint that the «`PointCut`» stereotype can only be applied to features of classes that are stereotyped «`Aspect`». `PointCut` is an abstract meta-class: This stereotype cannot be applied to the

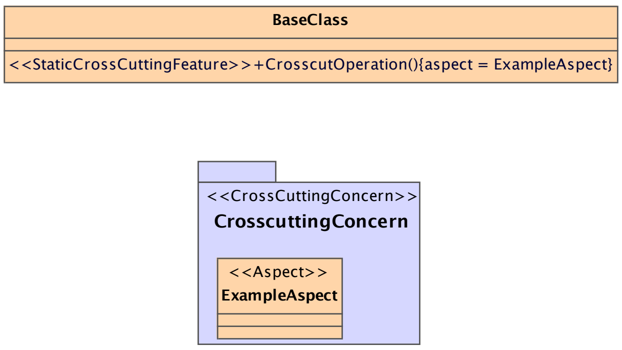


Figure 3: Alternate application of Static Crosscutting Features

attributes of an aspect; only its non-abstract sub-classes, such as `CallPointCut` or `ExecutionPointCut` can.

Rather than specifying the type and AspectJ textual declaration of pointcuts as attributes on `PointCut`, we subclass the `PointCut` meta-class to allow different attributes to be modelled for different pointcuts.

`OperationPointCut` is a superclass to describe pointcuts that select operation-related join points. Hence, this meta-class has a multi-valued attribute `operation` for this purpose, whose data type is the UML meta-class `Operation`. Because UML does not distinguish between operations and constructors, both `InitializationPointCut` and `PreInitializationPointCut` are subclasses of `OperationPointCut` and inherit the `operation` attribute.

`TypePointCut` is a superclass to describe pointcuts that select type-related join points. Therefore, it contains an ordered, multi-valued attribute `Type`, whose data type is the UML meta-class `Type`.

`AdviceExecutionPointCut` describes a pointcut that selects all advice execution.

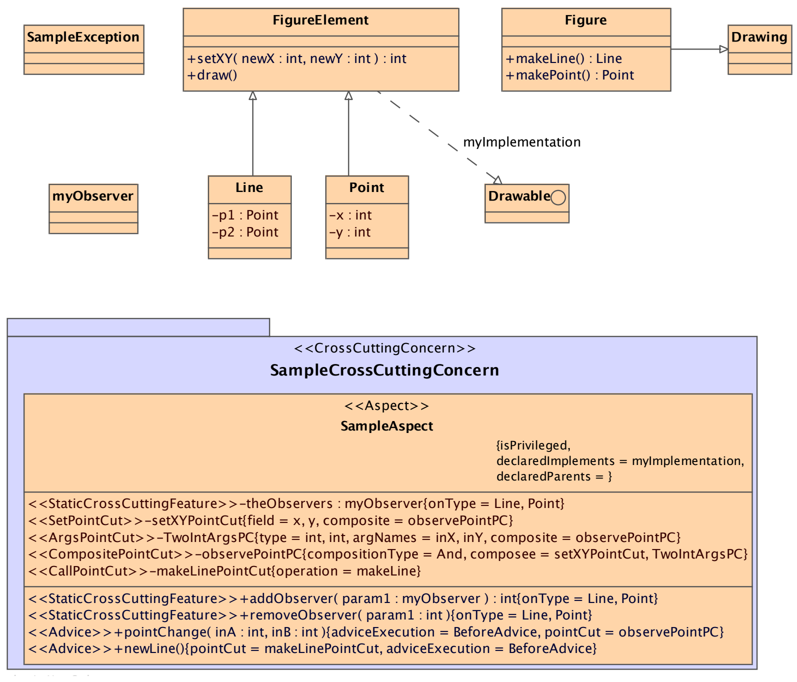
`PointCutPointCut` is a superclass for those types of pointcuts that select another pointcut. Hence, it is associated with the meta-class `PointCut` to specify the selected pointcuts.

`PropertyPointCut` is a superclass of those types of pointcuts that select fields. Therefore, it possesses a multi-valued attribute with data type `Property`.

`ContextExposingPointCut` is an abstract superclass of those types of pointcuts that can expose context in an advice. It contains an ordered, multi-valued attribute `argNames` that holds the names of the exposed arguments. This collection is ordered, so that the corresponding type can be discerned from the collection `type` specified for the `TypePointCut` meta-class.

In AspectJ, pointcuts can be composed of primitive pointcuts. Therefore, we have introduced the meta-class `CompositePointCut`, which is itself a subclass of `PointCut`, but also associated with `PointCut` to express those pointcuts of which it is a composite. The attribute `compositionType` specifies the boolean operator used for the composition.

We have chosen to make all references to joint points that are selected by pointcuts multi-valued (the `operation`, `field`, and `type` attributes on `OperationPointCut`, `PropertyPointcut` and `TypePointCut` respectively) to reduce the complexity of the resulting model. The alternative would



SampleException
FigureElement
+setXY(newX : int, newY : int) : int
+draw()
Figure
+makeLine() : Line
+makePoint() : Point
Drawing
myImplementation
myObserver
Line
-p1 : Point
-p2 : Point
Point
-x : int
-y : int
Drawable
<<CrossCuttingConcern>>
SampleCrossCuttingConcern
<<Aspect>>
SampleAspect
{isPrivileged,
declaredImplements = myImplementation,
declaredParents = }
<<StaticCrossCuttingFeature>>-theObservers : myObserver{onType = Line, Point}
<<SetPointCut>>-setXYPointCut{field = x, y, composite = observePointPC}
<<ArgsPointCut>>-TwoIntArgsPC{type = int, int, argNames = inX, inY, composite = observePointPC}
<<CompositePointCut>>-observePointPC{compositionType = And, composee = setXYPointCut, TwoIntArgsPC}
<<CallPointCut>>-makeLinePointCut{operation = makeLine}
<<StaticCrossCuttingFeature>>+addObserver(param1 : myObserver) : int{onType = Line, Point}
<<StaticCrossCuttingFeature>>+removeObserver(param1 : int){onType = Line, Point}
<<Advice>>+pointChange(inA : int, inB : int){adviceExecution = BeforeAdvice, pointCut = observePointPC}
<<Advice>>+newLine(){pointCut = makeLinePointCut, adviceExecution = BeforeAdvice}

(a) Base-model and crosscutting concern

```
package SampleCrossCuttingConcern;

privileged aspect SampleAspect {
    declare parents: Figure extends Drawing;
    declare parents: FigureElement implements Drawable;

    pointcut setXYPointCut (): (
        set(private int Point.x) ||
        set(private int Point.y));

   pointcut TwoIntArgsPC (int inX, int inY):
        args(inX, inY);

   pointcut observePointPC (int inX, int inY): (
        (set(private int Point.x) ||
         set(private int Point.y)) &&
        args(inX, inY));

   pointcut makeLinePointCut (): (
        call(public Line Figure.makeLine ()) );

   before(int inA, int inB) throws SampleException
                           : observePointPC(inA, inB ) {}
   before() : makeLinePointCut() {}

    private myObserver Line.theObservers;
    private myObserver Point.theObservers;
    public int Line.addObserver (myObserver param1) {};
    public int Point.addObserver (myObserver param1) {};
    public void Line.removeObserver (int param1) {};
    public void Point.removeObserver (int param1) {};
}
```

(b) Generated AspectJ code

Figure 4: Application of AspectJ profile

force the modeller to use pointcut composition using logical ”or”. When multiple features are specified for pointcuts, e.g. multiple values of the `operation` attribute of a `ExeuctionPointCut` instance, the assumption during code generation (Sect. 5) is that they are composed using logical ”or”.

Because pointcuts are used by advices, the meta-class `PointCut` is associated with the meta-class `Advice`.

4. PROFILE APPLICATION

We show an application of the proposed profile as proof of concept (Figure 4) and to identify benefits and shortcomings of the proposed UML extension. Rather than using a complex case study, we show a simple example to demonstrate the use of the profile’s during modelling.

Recall that in UML, meta-classes that extend existing meta-classes become stereotypes, and attributes of extending meta-classes become tags.

Crosscutting concerns become packages that are stereotyped «`CrossCuttingConcern`» and the aspects of this crosscutting concern are classes that are stereotyped «`Aspect`», contained in the package. The `isPrivileged` attribute of the meta-class `Aspect` becomes the tag `isPrivileged` of the stereotype «`Aspect`». In this example, the aspect declares a generalization and an interface realization relationship. Because of the meta-model integration, the values of the `declareParents` and `declaredImplements` tags are the relationships elements specified in the model. The UML `generalization` meta-class is not a subclass of the `NamedElement` meta-class, so that no name is shown for the value of the `declareParents` tag, but the interface realization dependency between `FigureElement` and `Drawable` is named, and this name appears as the value of the `declaredImplements` tag of the aspect. As Figs. 5 and 6 show, these are not textual specifications but refer to actual model elements.

Pointcuts are stereotyped attributes of the aspect, because they are defined as meta-class extensions of the `StaticFeature` meta-class. For example, the `setXYPointCut` attribute is stereotyped as a «`SetPointCut`». Its meta-class attribute `field` becomes a tag that provides a list of fields selected by this pointcut. The values of the fields are the attribute elements in the model (the CASE tool does not show the fully qualified name). Because of the meta-model integration, the CASE tool allows selection of the appropriate model elements as values, shown in Fig. 5. The figure shows the fully qualified names of the model elements that are the values of the tag `field`. Fig. 6 shows that the values of the tags can be picked out from the actual model elements. An example of a pointcut that exposes context variables is given with the `TwoIntArgsPC` example, selecting all operations taking two arguments of type `int`.

Static crosscutting features can be either atttributes or operations. Examples of both are shown as stereotyped «`StaticCrossCuttingFeature`». The meta-level attribute `onType` of the meta-class `StaticCrossCuttingFeature` becomes, on the model level, a tag with (multiple) values referencing the classes of the model. For example, the aspect introduces a public operation `addObserver (myObserver) : int` on the types `Line` and `Point`.

Advices are operations that are stereotyped «`Advice`». The `Advice` meta-class is associated with the `Pointcut` meta-class. Therefore, each advice in the aspect advises a pointcut, specified as the value of the tag `pointCut`.

5. CODE GENERATION

Because the model is compliant with standard UML XMI format and is fully specified in terms of the meta-model,

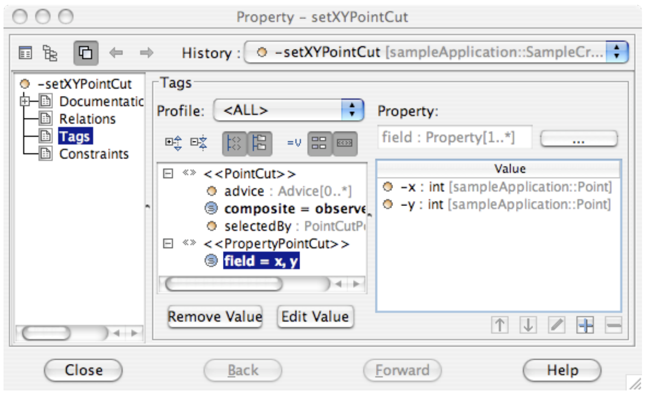


Figure 5: Tag values referring to model elements

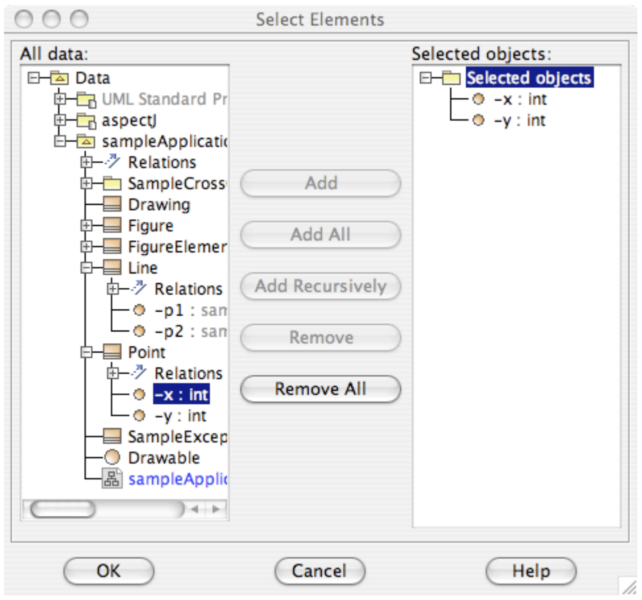


Figure 6: Selecting model elements as tag values

code can easily be generated. As a proof-of-concept, a brief XSLT has been implemented that generates valid AspectJ code (Fig. 4). Existing CASE tools already support code generation for the traditional parts of the model, so that the XSLT only generates code for the aspect. To give an indication of the complexity of the transformation, code generation is implemented in approx. 580 lines of XSLT code. Most of the complexity in the transformation stems from ensuring robustness. The XSLT is available from the author.

The code generation currently relies on the modeller to develop models that are valid representations of AspectJ. Some examples of this are the following:

The modeller must ensure that the signature of an `Advice` matches the context exposed on any referenced `ContextExposingPointCut`.

Another example of this onus on the modeller is the choice between a return parameter and a thrown exception on an advice. The XSLT will generate `after ... returning` when a return parameter is included in the advice signature, and will generate `after ... throwing` when a raised exception is modelled for the advice. However, it is valid in UML to model both.

While `TargetPointCut` and `ThisPointCut` have, in the interest of a simple meta-model, been modelled as subclasses of `TypePointCut`, they, in contrast to the other subclasses of `TypePointCut`, should only refer to a single type. When generating code, additional type references in the model are ignored.

Combining context-exposing primitive point cuts using multiple boolean operations can lead to very complex structures with no easily discernible signature. In the current implementation, the pointcut signature is generated from the signatures of the primitive `ContextExposingPointCuts` in a `CompositePointCut` using simple union. It is up to the modeller to ensure that this transforms to valid AspectJ code.

When an advice signature contains a return parameter, the XSLT will interpret this parameter depending on the value of the `adviceExecution` tag. For a `BeforeAdvice`, the return parameter type is ignored, for an `AroundAdvice` it is interpreted as the type of the value returned by the advice (generating "`type around(...):`"), while for an `AfterAdvice` it is interpreted as the type of the value returned by the operation (generating "`after(...) returning (type):`").

These examples of potential pitfalls when generating code highlight the need for future work to include OCL-based constraint specifications as part of the profile. Such constraints, if enforced, would be able to significantly reduce the complexity of the code generation. However, current UML modelling tools lack the ability to enforce OCL specifications.

6. DISCUSSION

We have shown an AspectJ profile for UML which, in contrast to previous work, is based completely on the existing UML meta-model, employing standard UML extension mechanism. This section discusses strengths and weaknesses of the proposal.

From a theoretical perspective, the strength of this proposal is a complete specification of AspectJ in UML. The model completely specifies AspectJ in terms of the UML meta-model and does not rely on textual descriptions or annotations that must be parsed for model application or verification. To our knowledge, this is the first complete proposal.

From a practitioner's perspective, using the lightweight, meta-model based extension mechanisms of UML 2.0 makes the theoretically important AspectJ meta-model practically useful as a profile. The profile can be used with existing, commercially available UML CASE tools[1]. Aspects can be exchanged using UML XMI model interchange mechanism and applied to both new and existing UML models. The modular way in which UML 2.0 allows profiles to be exchanged and applied means that AspectJ model extensions can be applied to existing UML models, just as AspectJ extensions are woven into existing Java software.

However, some words of caution are in order. The lack of pattern based, textual specification implies that each AO-feature refers to a specific model element that must be explicitly specified by the modeller (Figs. 5, 6). This is in contrast to the AspectJ language where patterns are used

[1]It has been implemented in the MagicDraw tool from No-Magic, Inc.

to select join points for pointcuts. The power of pattern specifications is not available in UML. Having to explicitly specify each pointcut requires that the modeller be aware of the complete base-system model. This also is in contrast to AspectJ where AO-features can be specified using pattern expressions without full knowledge of the specific join points or types selected by a pattern. However, this type of pattern-based specification, while convenient, also opens the door to inadvertent selection of unintended join points. In this respect, the explicit specification required by the presented profile is safer and the meta-model integration allows easier model checking and verification. Moreover, if patterns were to be specified using textual attributes in the UML model, special tools would be required to resolve such specifications on the model level, e.g. as part of model-level weaving. This would preclude the use of commercially available CASE tools for AspectJ modelling.

While one may argue that explicit specification of all AO-features creates a model almost as complex as if the crosscutting functionality had been included using non-aspect methods, the use of the proposed profile retains the main advantage of AO-modelling, namely that of modularization and encapsulation of crosscutting concerns.

The present work can be extended in multiple directions in future work. First, it does not yet fully take into account generics and annotations in Java 5 and 6. UML has the `TemplateableElement` concept and is therefore able to express Java generics. Java annotations may be modelled as stereotypes in UML. While the proposed profile can be applied to stereotyped and templated model elements, the code generating XSLT transformation needs to be extended to develop corresponding Java 5 code.

Second, in the context of the model driven architecture (MDA) process [13], two extensions can be developed. UML profiles can be developed for other aspect-oriented languages, such as Aspect#, to allow the development of platform specific models ("PSM" in MDA terminology). The aspect-oriented features can also be abstracted into a language-agnostic UML profile for generic AOM (platform independent models, "PIM" in MDA terminology). Transformations can then be developed to transform the language-agnostic aspect oriented models ("PIM") into language specific aspect-oriented models ("PSM") and from there to code.

Third, OCL constraints can be developed to ensure the validity of the models. For example, the signature of advices needs to match the context exposed by `ContextExposing-PointCuts`, so that valid AspectJ code is ensured.

Finally, usability studies need to be conducted. In this context, it is also feasible to explore the impact of various design decisions for this profile, e.g. textual specification of join points versus the present meta-model based specification, or the modelling of static crosscutting features with the aspect as presented here, or with the base model element.

7. REFERENCES

[1] O. Aldawud, T. Elrad, and A. Bader. A UML profile for aspect oriented modeling. In *Proceedings of OOPSLA 2001*, 2001.

[2] O. Aldawud, T. Elrad, and A. Bader. UML profile for aspect-oriented software development. In *Proceedings of the AOM workshop at AOSD, 2003*, 2003.

[3] E. Barra, G. Genova, and J. Llorens. An approach to aspect modelling with UML 2.0. In *Proceedings of the AOM workshop at AOSD, 2004*, 2004.

[4] M. Basch and A. Sanchez. Incorporating aspects into the UML. In *Proceedings of the AOM workshop at AOSD, 2003*, 2003.

[5] C. Chavez and C. Lucena. A metamodel for aspect-oriented modeling. In *Proceedings of the AOM with UML workshop at AOSD, 2002*, 2002.

[6] L. Fuentes and P. Sanchez. Elaborating UML 2.0 profiles for AO design. In *Proceedings of the AOM workshop at AOSD, 2006*, 2006.

[7] I. Groher and S. Schulze. Generating aspect code from UML models. In *Proceedings of the AOM workshop at AOSD, 2003*, 2003.

[8] J. Grundy and R. Patel. Developing software components with the UML, Enterprise Java Beans and aspects. In *Proceedings of ASWEC 2001, Canberra, Australia*, 2001.

[9] W. Harrison, P. Tarr, and H. Ossher. A position on considerations in UML design of aspects. In *Proceedings of the AOM with UML workshop at AOSD, 2002*, 2002.

[10] J.-M. Jezequel, N. Plouzeau, T. Weis, and K. Geihs. From contracts to aspects in UML design.

[11] M. Kande, J. Kienzle, and A. Strohmeier. From AOP to UML - a bottom-up approach. In *Proceedings of the AOM with UML workshop at AOSD, 2002*, 2002.

[12] F. Mostefaoui and J. Vachon. Formalization of an aspect-oriented modeling approach. In *Proceedings of Formal Methods 2006, Hamilton, ON*, 2006.

[13] Object Management Group. *MDA Guide*, June 2003. Document omg/2003-06-01.

[14] Object Management Group. *Unified Modeling Language: Superstructure*, Aug. 2005. Document formal/05-07-04.

[15] R. Pawlak, L. Duchien, G. Florin, F. Legond-Aubry, L. Seinturier, and L. Martelli. A UML notation for aspect-oriented software design. In *Proceedings of the AOM with UML workshop at AOSD, 2002*, 2002.

[16] A. Reina, J. Torres, and M. Toro. Towards developing generic solutions with aspects. In *Proceedings of the AOM workshop at AOSD, 2004*, 2004.

[17] D. Stein, S. Hanenberg, and R. Unland. Designing aspect-oriented crosscutting in UML. In *Proceedings of the AOM with UML workshop at AOSD, 2002*, 2002.

[18] H. Yan, G. Kniesel, and A. Cremers. A meta model and modeling notation for AspectJ. In *Proceedings of the AOM workshop at AOSD, 2004*, 2004.