JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2004

Vol. 6, No. 7, Special Issue: Aspect-Oriented Modeling, August 2007

A Meta-Level Specification and Profile for AspectJ in UML

Joerg Evermann, School of Information Management, Victoria University of Wellington, Wellington, New Zealand

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, the resulting meta-model is also a UML profile for supporting AspectJ modelling in UML. In contrast to previous work, this profile offers complete meta-level integration of all AspectJ concepts. Moreover, the use of standard XMI based modelling allows the use of the profile in commercially available CASE tools and supports easy code generation.

1 INTRODUCTION

The identification and separation of cross-cutting concerns in software design has led to the the widespread use of aspect-oriented programming (AOP) techniques. While AOP has matured and gained widespread industry support, there is little support for separation of cross-cutting concerns on the software model level. Aspect-oriented modelling (AOM) techniques are intended to fill this gap.

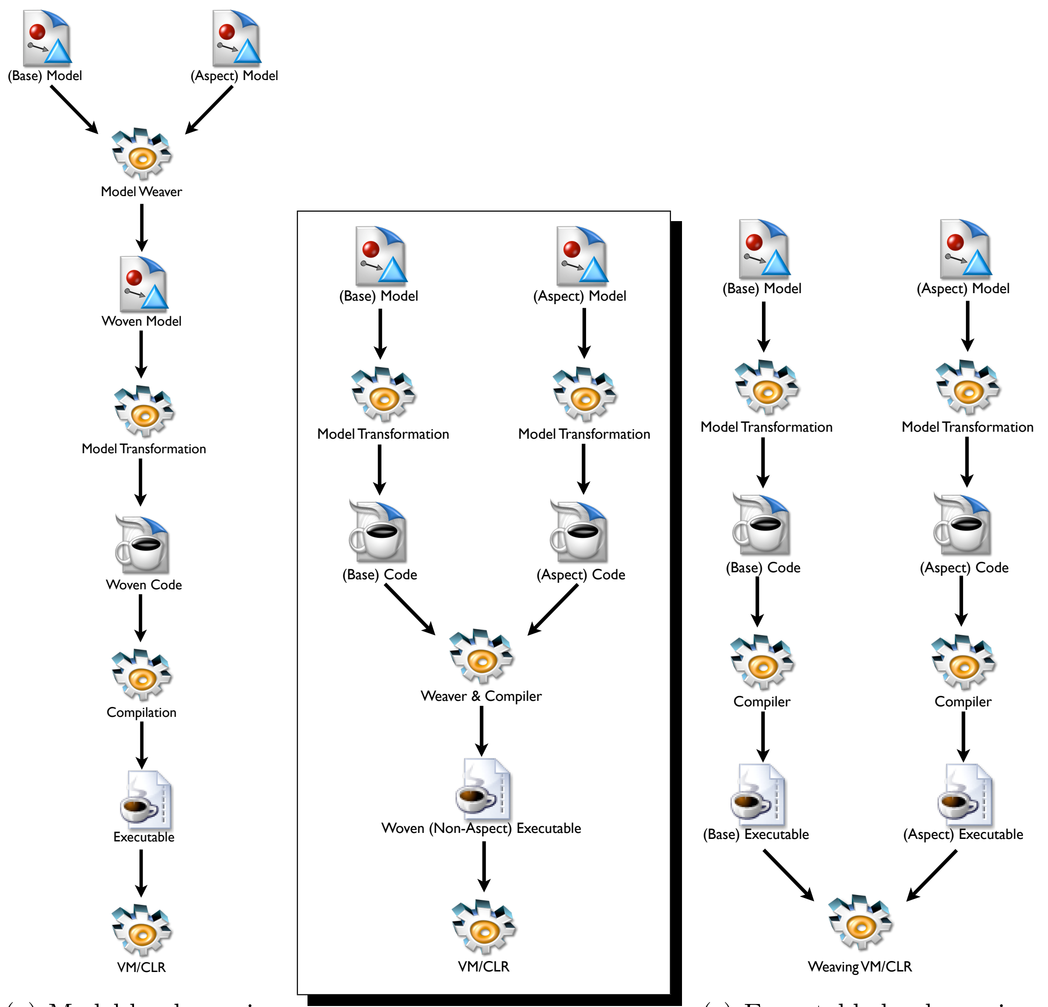
Aspect-Oriented Modelling and Positioning of This Work

AOM approaches can be distinguished along two orthogonal dimensions: the level of weaving and the symmetry of the approach (Fig. 1). The first dimension, the level of weaving, indicates that level of abstraction where separation of cross-cutting concerns is given up and weaving of concerns (aspects) occurs. In the case of model-level weaving, cross-cutting concerns are separately modelled, then woven together to yield a non-aspect-oriented model, which is then transformed to non-aspect-oriented code (Fig. 2a). In the case of code-level weaving, the separation of concerns is maintained to the level of code. Aspect-oriented models are transformed to aspect-oriented code, and the weaving is done by an aspect compiler on the code (Fig. 2b). Finally, recent developments have led to the idea of executable-level (runtime) weaving, where the separation of concerns is maintained on the executable (e.g. JVM or CLR bytecode) level and weaving occurs only at run-time (Fig. 2c).

Along the symmetry dimension, AOM approaches may be categorized as symmetric or asymmetric. For asymmetric AOM approaches, there is a designated base

	Symmetry	
Level of weaving	Symmetric Model weaving	Asymmetric Model weaving
	Symmetric Code weaving	Asymmetric Code weaving
	Symmetric Executable weaving	Asymmetric Executable weaving

Figure 1: Dimensions of Aspect-Oriented Modeling



(a) Model-level weaving (b) Code-level weaving (c) Executable-level weaving

Figure 2: Levels of weaving

model into which cross-cutting concerns (aspects) are woven. In the symmetric case, there is no such designated base system and the 'base system' is just one concern among many that is woven together.

Using these dimensions, the work presented here is positioned in the highlighted sector of Figure 1 and highlighted in Figure 2b: We present a technique to create aspect-oriented models that are converted to aspect-oriented code, which can be woven by an aspect-oriented compiler. We also make a clear distinction between the base-system and the cross-cutting concerns.

The focus on code-level weaving instead of model-level weaving is useful for software development projects where the system behaviour is not specified in UML behavioural diagrams. Many software development projects use only UML class diagrams [10], thereby ruling out model-level weaving of behaviour. For example, agile methods such as XP (eXtreme Programming) [6] reject elaborate model construction and eschew the use of MDA (model-driven-architecture).

Motivation and Goal

While aspect-oriented programming (AOP) is rapidly maturing and industrial-strength tools such as AspectJ [21] are quickly becoming widely applied *de-facto* standards, there is no standards-based approach to AOM that is supported by commercial modelling tools. UML [25], the *de-facto* standard software design language upon which many modelling tools are based, 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-oriented modelling facilities. In UML 2.0, these extension mechanisms allow the definition of UML profiles by means of meta-models.

In this paper, we present a meta-model of the AspectJ language [21]. Using the powerful extension mechanisms in UML 2.0, this meta-level model *is* a UML profile.

In the context of Model-Driven-Architecture (MDA), our proposal is situated as shown in Figure 3. Our profile allows the specification of a platform-specific model (PSM), namely one that is specific to the Java and AspectJ platform. We also offer a translation to code, as indicated by the circled elements in Figure 3.

The presented profile is not a generic aspect-oriented modelling extension. The conceptual differences between different aspect implementations, e.g. AspectC++, Aspect# and AspectJ are substantial and cannot readily be captured in a single meta-model. We have focused on providing a UML extension to support AspectJ because of the maturity of the development of AspectJ and its wide-spread industrial use [21].

Generic aspect-oriented profiles may be used for specification of a platform-independent model (PIM), which in MDA forms the basis of the platform-specific model (PSM) (Fig. 3). A generic aspect profile and platform specific aspect profiles for languages other than AspectJ are beyond the scope of this paper.

Our approach as outlined here offers the following advantages over previous proposals (discussed in detail in Section 2). First, it is supported by UML 2.0 compliant modelling tools. The extension requires no special software support and allows aspect modelling to be used within existing, mature software tools. For example, the

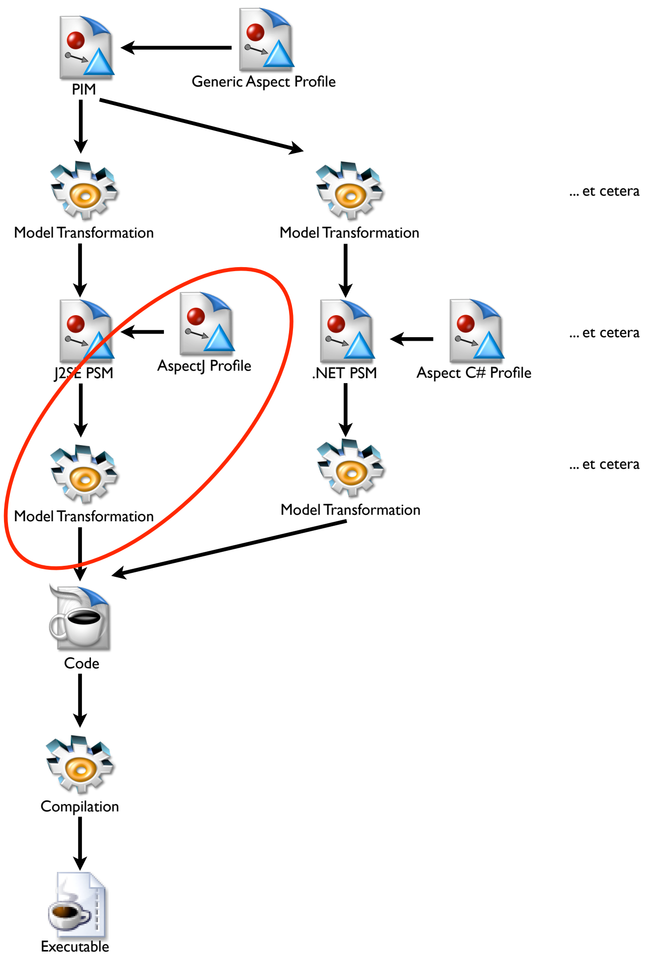


Figure 3: Use of profile within MDA-based development

work described in this paper was developed using the commercially available tool MagicDraw, version 11.5. This contrasts with earlier proposals, which are not all based on profiles and extend UML either by introducing new meta-model classes, or new notation elements, or both [13, 31]. 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 is applied to, can be exchanged between different MOF (Meta-Object-Facility) compliant UML modeling tools.

Third, the proposal allows all aspect-related concepts to be specified in meta-model terms. Hence, no textual specification of special keywords is necessary. This means the models can be easily manipulated or verified, without requiring the parsing of keywords or other textual specifications by special tools.

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

The remainder of this paper is structured as follows. Section 2 presents related work. Section 3 represents the main contribution of this work and discusses the proposed AspectJ meta-model. This is followed by a brief example of the use of this profile in Section 4. We describe the code-generation capabilities that have been developed for this profile (Sec. 5) before concluding the paper with an outlook to future research (Sec. 6).

2 RELATED WORK

An overview of some of the prior work for modelling aspects in UML is presented in [27]. The early work presented at the AOM 2002 workshop 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 cross-cutting concerns that is central to AOP.

Initial work presented in [1] proposed the specification of aspects as stereotypes on classes and was later extended to include advice and pointcut specification [2]. It models cross-cutting associations to show which aspect features relate to which base-model elements. It is a generic profile that captures only few of the AspectJ extensions.

The proposal in [26] 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 [18], however without being fully defined in UML meta-model terms. Aspects as well as pointcuts are stereotypes on UML classes. Pointcuts can be associated with UML classes by means of stereotyped ”binding” associations.

An earlier profile for AspectJ [28] proposes that messages in collaborations are join points. This profile represents advices and pointcuts as stereotyped operations, introduction of fields or methods is modelled as templated collaborations, and the connection to the base features is made via dependencies in the model. Similarly, the proposal in [5] uses textual specification of pointcuts, rather than being based on (meta-)model elements. Later extensions to this in [11] are similar to our proposal in that aspects are stereotyped classes. However, 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 [4], 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 [13] introduces new UML meta-classes and therefore requires specialized tools for their support. A meta-model for generic AOP is offered in [7], 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 [31]. 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 [16]. A translation of aspect UML to object-oriented Petri-nets is described in [23] but is limited to pointcuts around method calls. Weaving on the model level is presented in [17] as part of work on design-by-contract. Code generation from aspect-extended UML models is presented in [12], who opt against the XMI based method proposed in this paper and instead use custom tool extensions.

In summary, much of the existing work on AOM profiles for UML is either based on older UML versions, not well integrated on the meta-model level, incomplete with respect to AspectJ, or requires specific tool support. The present proposal addresses these gaps in the prior work.

3 ASPECT UML EXTENSION

The main point of distinction of the present 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 model elements, 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 cross-cutting 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. 13, 14).

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. 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 [25]. Visually, this is shown with the extended class in square brackets. Attributes that are modelled on stereotype meta-classes will translate to tags when the profile is applied [25]. Similarly, values of stereotype attributes will become values of tags when the profile is applied [25]. This extension mechanism in UML 2.0 is therefore a 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. We show the complete model in Figure 11 below.

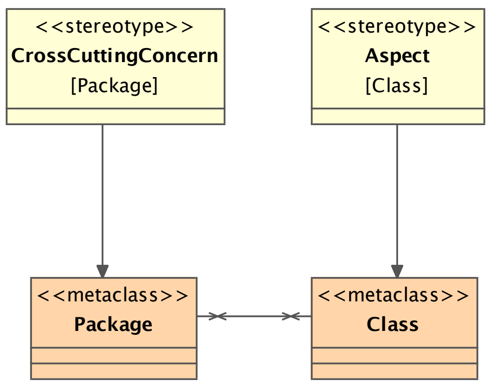


Figure 4: Cross cutting concern as package extension

CrossCuttingConcern We introduce the meta-class `CrossCuttingConcern` as a way of grouping related aspects. While cross-cutting concerns are not directly specifiable in AspectJ, being able to package related aspects is important for modularization and reuse [8, 9, 3]. `CrossCuttingConcern` extends the UML meta-class `Package`, because a cross-cutting concern contains aspects in the same way as packages contain classes (Fig. 4). Providing a stereotyped package to contain aspects also allows the packaging of supporting classes and supporting interfaces for aspects in these packages to enhance modularization. For example, an aspect `A` may depend on classes `C1` and `C2` and the `CrossCuttingConcern` allows these to be packaged together. 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` (Fig. 5). This makes the `Aspect` meta-class a stereotype on the UML `Class` construct.

Recall that this proposal is positioned in asymmetric AOM (Fig 1). Hence, elements of the cross-cutting concern model or models must remain separated from base-model elements. The following constraint ensures this by requring that classes that are stereotyped «`Aspect`» are only packaged in packages that are stereotyped as «`CrossCuttingConcern`».

```
context Aspect inv:
  package.oclIsKindOf(CrossCuttingConcern)
```

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

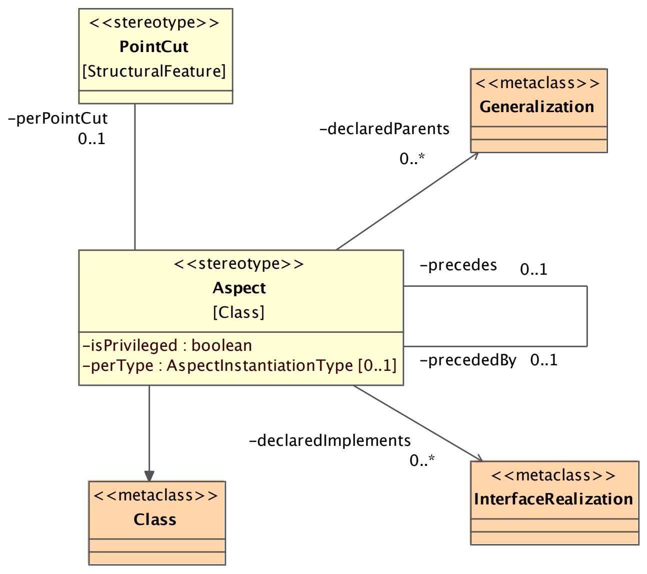


Figure 5: Aspect as class extension

is privileged, i.e. whether its advices can access private or protected members of the class they advise.

A multi-valued attribute `declaredParents` allows the declaration of generalizations by the aspect ("`declare parents: I extends J`" in AspectJ). Because the aspects are a meta-level model element, 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. For exposition purposes, Figure 5 explicitly shows the meta-level connections modelled as associations. Figure 11 uses atttributes with appropriate datatypes for reasons of brevity.

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`. The values of this stereotype tag pick out interface realization model elements when this extension is applied.

Because aspects may be instantiated per pointcut, the attributes `perType` and `perPointCut` are used to specify the type of aspect instantiation and associated pointcut, if any. 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`. We add the following constraint to ensure that aspect instantiation is indicated either per type or per pointcut, but not for both:

```
context Aspect inv:
  perType -> size() > 0 xor
  perPointCut -> size() > 0
```

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.

The AspectJ specification states that aspects may extend classes or other aspects and implement interfaces, but that classes may not extend aspects. Consequently, we add the following constraint that ensures for all generalizations that the specific class of a general class that is an aspect is also an aspect:

```
context Generalization inv:
  general.oclIsKindOf(Aspect) implies specific.oclIsKindOf(Aspect)
```

Because aspects are extensions of the meta-class `Class`, dependencies between aspects can easily be defined. Moreover, visibility declarations and interfaces can be used to establish which parts of an aspect are visibile to or hidden from other aspects. This enables modellers to use aspect-composition frameworks, such as the one proposed in [19].

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 of the meta-class `BehavioralFeature` (Fig. 6).

In UML, the meta-class `BehavioralFeature` with subclasses such as `Operation` or `Reception` (of a signal) is associated with the `Behavior` meta-class, which is the superclass of `OpaqueBehavior`, `Activity`, `Interaction` and `StateMachine`, and specifies the actual behavior of a behavioral feature. Opaque behaviour denotes methods whose body is specified in a UML-external language (and thus is opaque as far as UML tools are concerned). This means that advice behavior can be specified either as a method, an activity, an interaction or a state machine. This mirrors a previous proposal 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`». In other words, for all instances of an advice, the classifier of the advice feature must be an aspect:

```
context Advice inv:
  allInstances.featuredClassifier.oclIsKindOf(Aspect)
```

The UML meta-model already associates operations with signatures. Hence, our extension does not need to model signatures for advices when the advice stereotype is applied to operations, a subclass of behavioral features. 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.

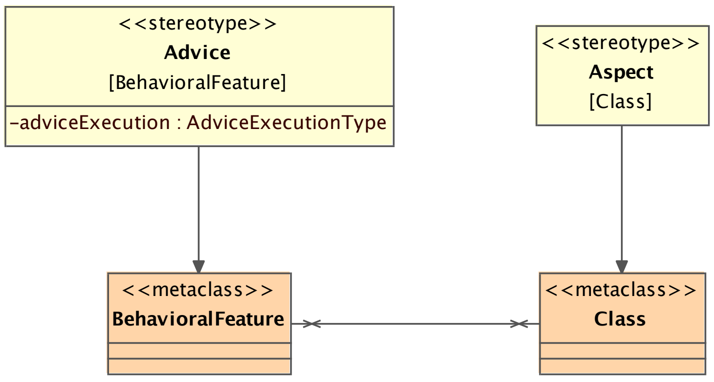


Figure 6: Advice as behavioral feature extension

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 determines whether an ”after” advice is ”after returning” or ”after throwing” by examining whether the signature of an operation that is stereotyped «`Advice`» contains a return parameter or a raised exception, respectively.

Static Crosscutting Features Aspects may introduce new features to existing classes and types. Because such cross-cutting 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` (Fig. 7). Because 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`». In other words, for all instances of an introduced cross cutting feature, the classifier of that instance must be an advice.

```
context StaticCrossCuttingFeature inv:
  allInstances.featuredClassifier.oclIsKindOf(Aspect)
```

To specify on which types the cross-cutting feature is to be introduced, the `StaticCrossCuttingFeature` meta-class possesses a multi-valued attribute `onType` whose data type is the UML meta-class `Type`. For exposition purposes, Figure 7 explicitly shows the meta-level connections modelled as an association. Figure 11 uses an atttribute with the appropriate datatype for reasons of brevity.

We have chosen to model cross-cutting features as owned by the aspect, rather than by the classifier they are introduced on. While this requires the extra meta-model attribute `onType`, it enforces the separation of base-model and cross-cutting concerns that is fundamental to AOP. In the alternative model, the meta-class `Aspect` would be associated with the meta-class `Feature` so that the aspect can pick

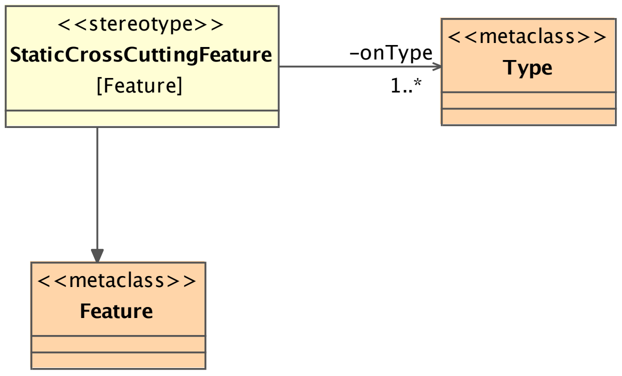


Figure 7: Static cross-cutting feature as feature extension

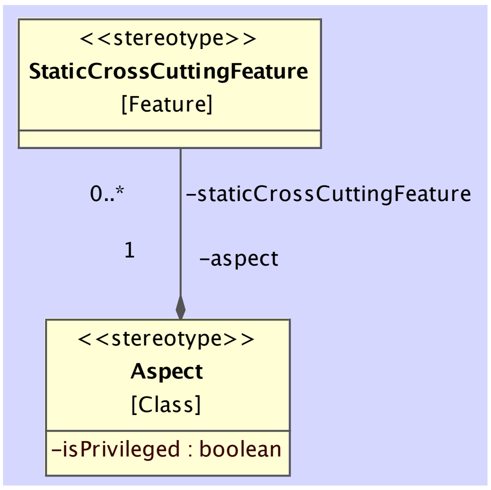


Figure 8: Alternate model for Static Crosscutting Features

out any feature owned by any classifier in the model (Fig. 8). The application of this alternate meta-model in Figure 9 shows that in this case the static cross-cutting 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, at least visually.

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`».

```
context PointCut inv:
  allInstances.featuredClassifier.oclsIsKindOf(Aspect)
```

`PointCut` is an abstract meta-class: This stereotype cannot be applied to the attributes of an aspect; only its non-abstract sub-classes, such as `CallPointCut` or

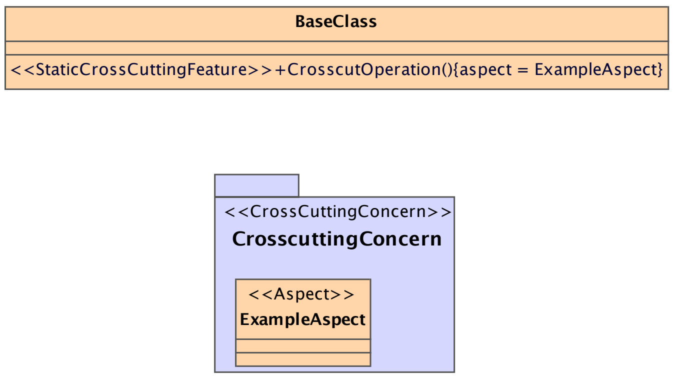


Figure 9: Alternate application of Static Crosscutting Features

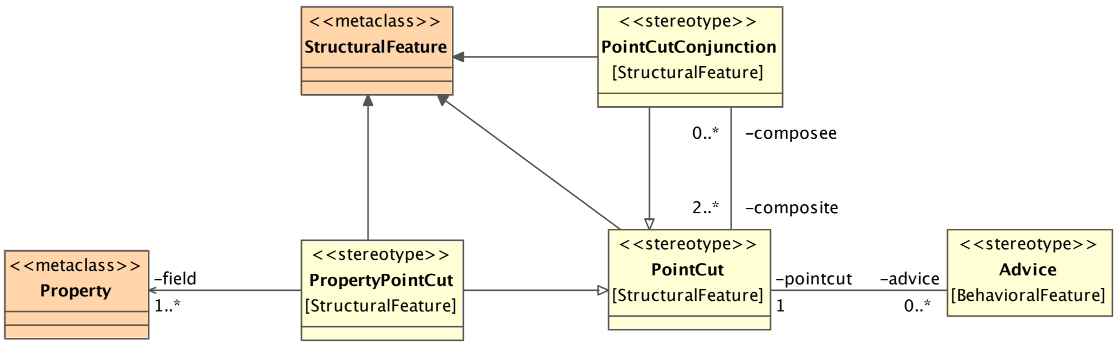


Figure 10: Pointcut meta-model excerpt

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. Figure 10 shows the meta-model excerpt for pointcuts and one subclass of pointcut, the property pointcut. Other subclasses of pointcuts are modelled analogously (Fig. 11).

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. The subclasses of this pointcut directly reflect AspectJ pointcut types and the reader is referred to the AspectJ specification or [21] for the precise semantics. 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. The subclasses of TypePointCut again directly reflect AspectJ pointcut types and the reader is again referred to the AspectJ specification or [21] for the precise semantics.

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 ordered collection type specified for the TypePointCut meta-class.

In AspectJ, pointcuts can be composed of primitive pointcuts. We therefore introduce three kinds of pointcut composition meta-classes: PointCutConjunction, PointCutDisjunction and PointCutNegation. These are modelled as separate sub-classes because the negation operation accepts only a single operand, while conjunction and disjunction require at least two. No ordering of the operands for conjunction or disjunction is necessary because these operations are associative and commutative. Figure 10 shows an excerpt including the PointCutConjunction meta-class. Disjunction and negation are modelled analogously (Fig. 11).

We have chosen to make all references to join 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 force the modeller to use pointcut composition using pointcut disjunction. 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 disjunction.

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

Figure 11 shows the complete profile developed in this section. In this representation the meta-classes that are extended are not shown explicitly but are shown in the class symbols in square brackets. In the above discussion we have shown associations to existing meta-classes for purposes of exposition and clarity. In the final representation, these associations have been modelled instead as attributes of classes with the appropriate datatypes.

4 PROFILE APPLICATION

We show an application of the proposed profile as proof of concept (Fig. 12) 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

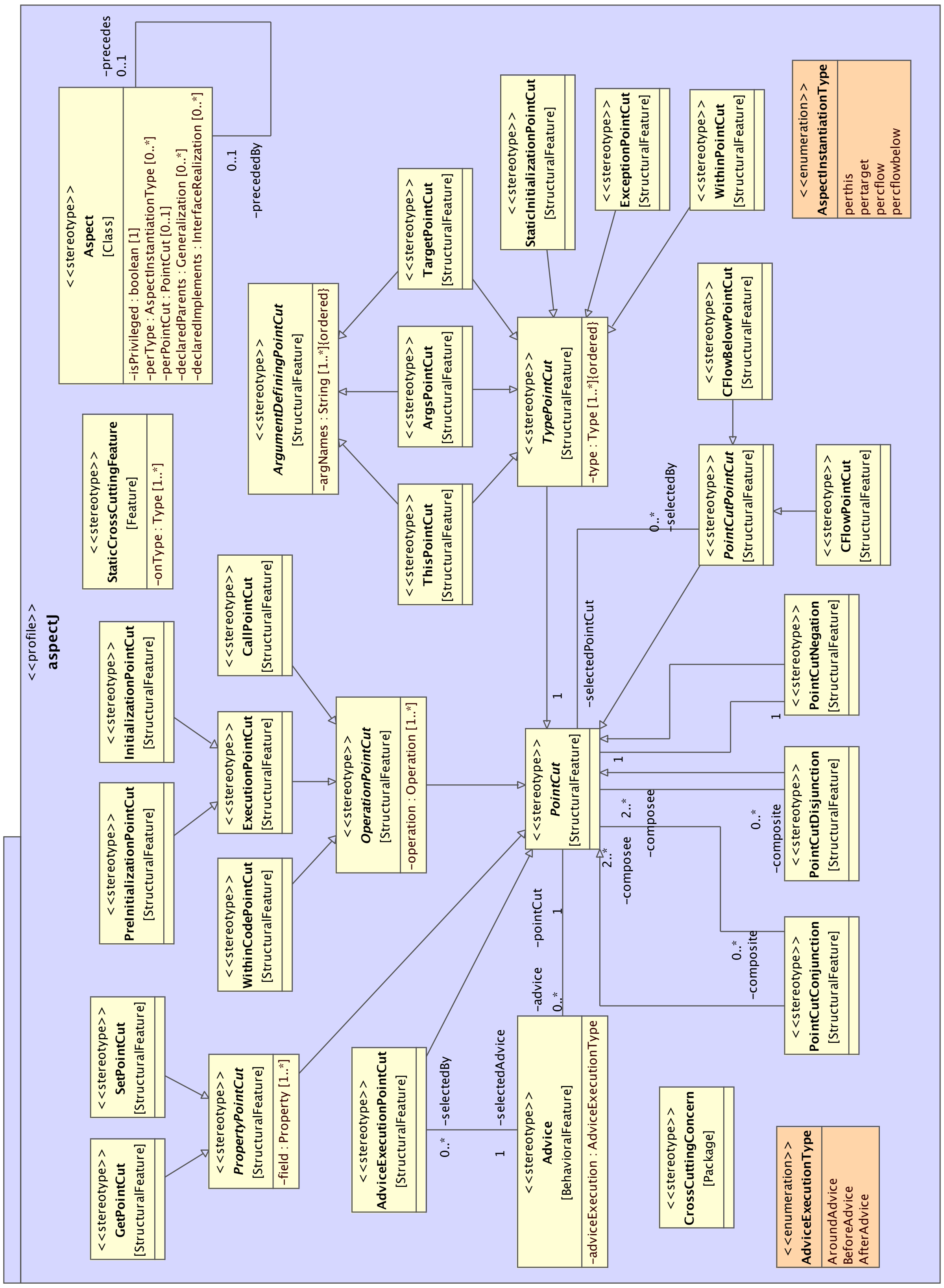

<<profile>>
aspectJ
<<stereotype>> Aspect [Class]
-isPrivileged : boolean [1]
-perType : AspectInstantiationType [0..*]
-perPointCut : PointCut [0..1]
-declaredParents : Generalization [0..*]
-declaredImplements : InterfaceRealization [0..*]
-precedes 0..1
0..1 -precededBy
<<stereotype>> StaticCrossCuttingFeature [Feature]
-onType : Type [1..*]
<<stereotype>> ArgumentDefiningPointCut [StructuralFeature]
-argNames : String [1..*]{ordered}
<<stereotype>> TargetPointCut [StructuralFeature]
<<stereotype>> ArgsPointCut [StructuralFeature]
<<stereotype>> ThisPointCut [StructuralFeature]
<<stereotype>> StaticInitializationPointCut [StructuralFeature]
<<stereotype>> ExceptionPointCut [StructuralFeature]
<<stereotype>> WithinPointCut [StructuralFeature]
<<enumeration>> AspectInstantiationType
perthis
pertarget
percflow
percflowbelow
<<stereotype>> TypePointCut [StructuralFeature]
-type : Type [1..*]{ordered}
<<stereotype>> CFlowBelowPointCut [StructuralFeature]
<<stereotype>> PointCutPointCut [StructuralFeature]
<<stereotype>> CFlowPointCut [StructuralFeature]
0..* -selectedBy
-selectedPointCut
1
<<stereotype>> CallPointCut [StructuralFeature]
<<stereotype>> InitializationPointCut [StructuralFeature]
<<stereotype>> ExecutionPointCut [StructuralFeature]
<<stereotype>> PreInitializationPointCut [StructuralFeature]
<<stereotype>> WithinCodePointCut [StructuralFeature]
<<stereotype>> OperationPointCut [StructuralFeature]
-operation : Operation [1..*]
<<stereotype>> PointCut [StructuralFeature]
<<stereotype>> PointCutNegation [StructuralFeature]
1
1
<<stereotype>> PointCutDisjunction [StructuralFeature]
2..* -composee
0..* -composite
<<stereotype>> PointCutConjunction [StructuralFeature]
2..* -composee
0..* -composite
-advice 0..*
-pointCut 1
<<stereotype>> SetPointCut [StructuralFeature]
<<stereotype>> GetPointCut [StructuralFeature]
<<stereotype>> PropertyPointCut [StructuralFeature]
-field : Property [1..*]
<<stereotype>> AdviceExecutionPointCut [StructuralFeature]
0..* -selectedBy
1 -selectedAdvice
<<stereotype>> Advice [BehavioralFeature]
-adviceExecution : AdviceExecutionType
<<stereotype>> CrossCuttingConcern [Package]
<<enumeration>> AdviceExecutionType
AroundAdvice
BeforeAdvice
AfterAdvice

Figure 11: AspectJ profile for UML

the profile's during modelling and to show the visual appearance of the model.

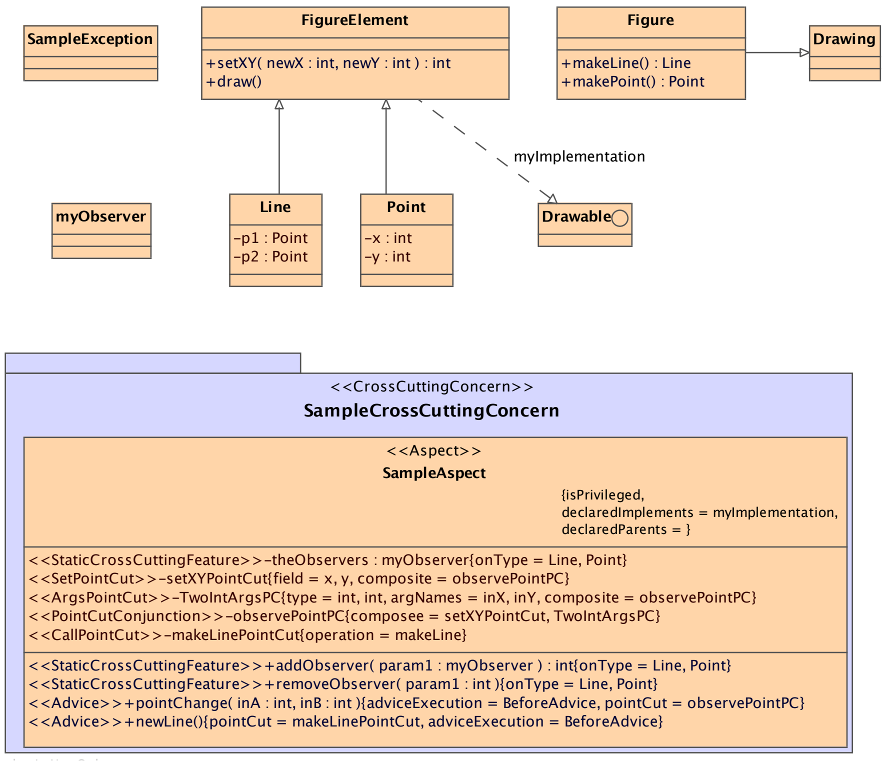
Recall that in UML, meta-classes that extend existing meta-classes become stereotypes, and attributes of extending meta-classes become tags.

Crosscutting concerns become packages that are stereotyped «CrossCutting-Concern» and the aspects of this cross-cutting 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 declaredParents 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 declaredParents tag, but the interface realization dependency between FigureElement and Drawable is named. This name appears as the value of the declaredImplements tag of the aspect. As Figs. 13 and 14 show, these are not textual specifications but refer to actual model elements. This means that XMI based tools can be used to manage and modify these features without requiring specialized parser extensions.

Pointcuts are stereotyped attributes of the aspect, because they are defined as meta-class extensions of the StaticFeature meta-class. For example, the setXY-PointCut 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 Figure 13. The figure shows the fully qualified names of the model elements that are the values of the tag field. Figure 14 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 cross-cutting 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.


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}
<<PointCutConjunction>>-observePointPC{compose = 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 cross-cutting 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 12: Application of AspectJ profile

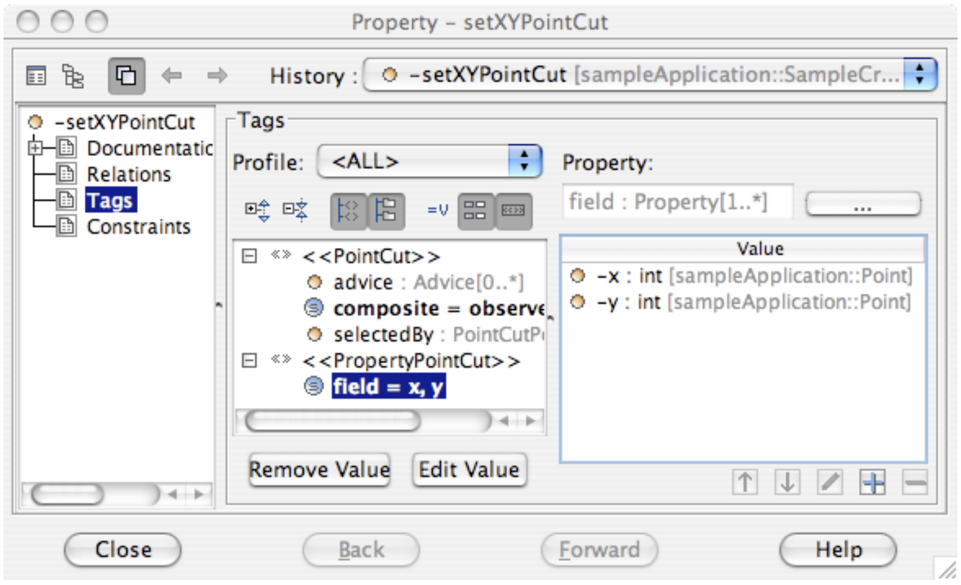


Figure 13: Tag values referring to model elements

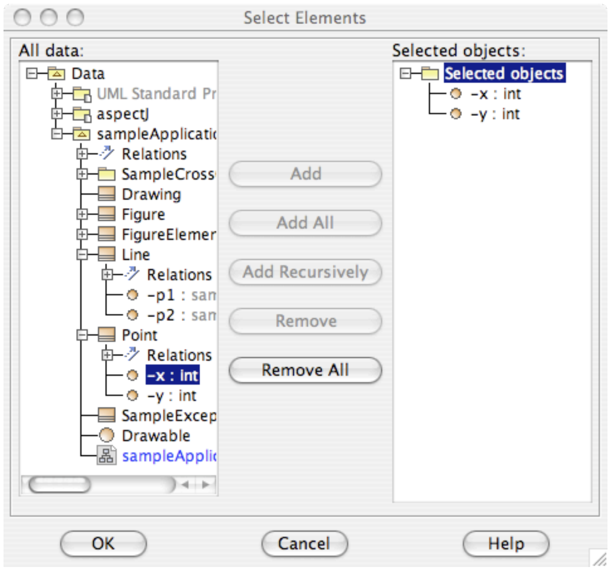


Figure 14: Selecting model elements as tag values

5 CODE GENERATION

Because the model is compliant with standard UML XMI format and is fully specified in terms of the meta-model, code can easily be generated. As a proof-of-concept, an XSLT has been implemented that generates valid AspectJ code (Fig. 12). Existing CASE tools already support code generation for the non-aspect-oriented parts of the model, so that the XSLT only generates code for classes stereotyped as «`Aspect`» within packages that are stereotyped «`CrossCuttingConcern`». The XSLT will generate method bodies for advice behavior that has been specified as a method body (opaque behavior) in the model. If advice behavior has been specified as an activity, interaction, or state machine, the XSLT will create only a method stub. The translation of those types of behavioral specifications to Java method bodies is clearly beyond the scope of this research.

While there is not enough space here to present the complete XSLT, 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 freely available from the author.

The code generation currently relies on the modeller to develop models that are valid representations of AspectJ. For examples, 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. To at least partially address this issue, we include the following constraint:

```
context Advice inv:
  raisedException -> size() > 0 xor
  ownedParameter -> select(op : op.direction='return') -> size() > 0
```

While `TargetPointCut` and `ThisPointCut` have, in the interest of a simple metamodel, 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. This can be addressed using OCL constraints:

```
context TargetPointCut inv:
  type.size() < 2 and
  argNames.size() < 2
context ThisPointCut inv:
  type.size() < 2 and
  argNames.size() < 2
```

Combining context-exposing primitive pointcuts 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 OCL-based constraint specifications as part of the profile. Such constraints, if enforced, can significantly reduce the complexity of the code generation. However, while this profile includes OCL constraints, most current commercial UML modelling tools lack the ability to enforce them.

6 DISCUSSION

We have shown an AspectJ profile for UML which, in contrast to previous work, is based entirely 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 and the MOF allow profiles to be exchanged and applied means that AspectJ model extensions can be applied to existing UML models, just as AspectJ extensions can be 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. 13, 14). This is in contrast to the AspectJ language where patterns are used 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. This problem is known as the fragile pointcut problem [14, 20] and is especially problematic when refactoring [22] the base system code, because pattern-based pointcut specifications depend strongly on the specific design of the base system. In this

[1]It has been implemented in the MagicDraw tool from NoMagic, Inc.

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. A future extension to this work may investigate whether the use of JPDD (joint point description diagrams) [29, 30, 15] can address this issue. JPDDs are used for describing joint point selection in pointcuts. Because JPDDs are a type of UML diagram, an integration with this proposal may be possible.

While one may argue that explicit specification of all AO-features creates a model almost as complex as if the cross-cutting 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 cross-cutting 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 [24] (Fig. 3), 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, to be used for platform independent models (”PIM” in MDA terminology). Transformations can then be developed to transform the language-agnostic aspect-oriented models (”PIMs”) into language specific aspect-oriented models (”PSMs”) and from there to code.

Third, while some OCL constraints are presented, others can be developed to further ensure the validity of the models. For example, the signature of `Advice`s needs to match the context exposed by `ContextExposingPointCut`s, so that valid AspectJ code is ensured.

Fourth, while the present work describes the use of this profile for code-level weaving (refer to Figure 1 in Sect. 1 for the positioning of this work), it is also conceivable that a code-generating XSLT can be be developed for model-level weaving. From a woven UML model, the regular code-generation capabilities of commercial UML tools can then be used to create ordinary Java code. However, while feasible, it would require essentially duplicating the existing AspectJ weaver on the UML model level.

Finally, usability studies need to be conducted. In this context, it is also possible 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 cross-cutting features with the aspect as presented here, or with the base model element.

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. Baniassad and S. Clarke. Theme: An approach for aspect-oriented analysis and design. In *Proceedings of the International Conference on Software Engineering, 2004.*, 2004.

[4] 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.

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

[6] K. Beck. *Extreme programming explained: embrace change.* Addison-Wesley, Reading, MA, 2000.

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

[8] S. Clarke and R. J. Walker. Composition patterns: an approach to designing reusable aspects. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 5–14, Washington, DC, USA, 2001. IEEE Computer Society.

[9] S. Clarke and R. J. Walker. Towards a standard design language for AOSD. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 113–119, New York, NY, USA, 2002. ACM Press.

[10] B. Dobing and J. Parsons. How the UML is used. *Communications of the ACM*, 49(5), 2006.

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

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

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

[14] K. Gybels and J. Brichau. Arranging language features for more robust pattern-based crosscuts. In *Proceedings of the 2nd international conference on Aspect-Oriented Software Development (AOSD)*, pages 60–69, 2003.

[15] S. Hanenberg, D. Stein, and R. Unland. From aspect-oriented design to aspect-oriented programs: Tool-supported translation of JPDDs into code. In *Proceedings of the 6th International Conference on Aspect-Oriented Software Development AOSD, Vancouver, Canada*, pages 49–62, 2007.

[16] 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.

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

[18] 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.

[19] M. Katara and S. Katz. A concern architecture view for aspect-oriented software design. *Software and Systems Modeling*, 2007.

[20] A. Kellens, K. Mens, J. Brichau, and K. Gybels. Managing the evolution of aspect-oriented software with model-based pointcuts. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 501–525, 2006.

[21] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications, Greenwich, UK, 2003.

[22] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, Feb. 2004.

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

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

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

[26] 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.

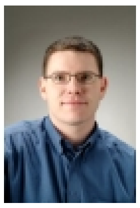
[27] A. Reina, J. Torres, and M. Toro. Towards developing generic solutions with aspects. In *Proceedings of the AOM workshop at AOSD, 2004*, 2004.

[28] 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.

[29] D. Stein, S. Hanenberg, and R. Unland. Query models. In *Proceedings of UML'04, Lisbon, Portugal*, pages 98–112, 2004.

[30] D. Stein, S. Hanenberg, and R. Unland. Expressing different conceptual models of joint point selections in aspect-oriented design. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development AOSD, Bonn, Germany*, pages 15–26, 2006.

[31] 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.



Joerg Evermann is a faculty member with the School of Information Management at Victoria University of Wellington. He received his PhD in MIS from the University of British Columbia. His research interests include requirements engineering and conceptual modeling, modeling issues in system analysis and software design, and the evaluation and improvement of modeling languages.