Modeling Pointcuts

Dominik Stein, Stefan Hanenberg, and Rainer Unland
Institute for Computer Science and Business Information Systems
University of Duisburg-Essen, Germany
{dstein | shanenbe | unlandR}@cs.uni-essen.de

Abstract

Modeling pointcuts, i.e., modeling the places where to crosscut and/or the conditions under which to crosscut, is a principal task in aspect-oriented modeling. It is a fairly independent design issue and can be accomplished separate from other modeling tasks such as modeling the crosscutting effects. Modeling pointcuts is basically about modeling selection queries. It requires novel modeling means. This paper gives a short overview on a new graphical approach to model pointcuts. It presents its semantics using OCL code. It presents its use in the early aspect phase, and it demonstrates its capabilities to capture the pointcut semantics of prevailing aspect-oriented programming techniques with help of examples.

1. Introduction

Aspect-Oriented Software Development (AOSD) is about encapsulating crosscutting concerns in aspects, and weaving these aspects together with other aspects and basic functionalities to a final aspect-enhanced program. When designing aspects or aspect-oriented software architectures, a principal task of aspect-oriented software developers is to investigate the relationships between aspects and their target artifacts. The software developer needs to contemplate about the aspect's assertions made on the final program as well as on the places and conditions at/under which the assertions apply to the program. Ultimately, the aspect-oriented weaver will take this information and enhance the final program accordingly. The key goal of aspect-oriented software development is to make reuse of aspects, i.e., crosscutting concerns, as easy as adjusting places and conditions of crosscutting to new programs and/or new requirements. To achieve this goal, aspect-oriented designers require means to specify such places and conditions of crosscutting from the early stages of architecture design.

Currently, aspect-oriented software development is greatly advancing on the implementation level. However, comprehensive design support is still in its infancy. Promising methodologies for requirements analysis, architecture design, and graphical visualizations are around (e.g., [13], [32], [35], [20], [30], [19], etc.); yet improvements are necessary to span the entire software life cycle and to cope with multiple aspect-oriented implementation techniques.

This paper deals with modeling and graphical visualization of places and conditions of crosscutting. Defining places and conditions at/under which an aspect affects the final program turns out to be a critical design activity in AOSD. Pointcut design in aspect-oriented architecture design is likewise crucial as interface design in conventional architecture design since both specify the connection points between different software artifacts of the architecture. Inaccurate pointcut definition may thus easily require fundamental redesign of the aspect and/or the architecture at a later stage in software development. Furthermore, lax definition of places and conditions of crosscutting quickly designates much more parts in the final program than was intended. Such will severely obstruct reasoning on the crosscutting effects of aspects and, in the ultimate, lead to unpredictable software comportment. Hence, it is essential that software designers are supported from the early stages of system design in identifying and documenting (!) the places and conditions at/under which an aspect crosscuts the final program.

Our work is focused on the Unified Modeling Language (UML) [28]. The UML is a powerful and broadly used modeling language for use case driven, architecture oriented, iterative, and incremental software development [12]. It may be used throughout the entire software development process [18] and may be used with different object-oriented implementation languages. Hence, the UML already provides much of what we are trying to achieve for aspect-oriented software development. It appears very appealing to exploit its capabilities for aspect-oriented software modeling.

Aspect-oriented modeling using the UML has been one of the core subjects of the ongoing series of workshops on Aspect-Oriented Modeling [1] [2] [3] [4]. The fruitful discussions and important insights given at these workshops have strongly influenced this work.

In this paper, we borrow the terms “join point” and “pointcut” from AspectJ [9] terminology. In difference to AspectJ, however, we contemplate on “join points” as “hooks where enhancements may be added” (cf. [14]) rather than as “principal points in the execution of a program” (cf. [10]). That means, we consider “join point” to refer to both points of crosscutting in the control flow as well as points of crosscutting in the class structure. We shall use the term “pointcut” to refer to a set of join points that possibly is attributed with conditions of crosscutting.

The remainder of this work is structured as follows: Section 2 discusses why pointcut modeling is a distinct design issue and why it must be given special care. We do this while first looking at the general case in aspect-oriented requirement engineering [30]. Afterwards, we reflect on the current aspect-oriented programming techniques. Section 3 presents our approach to model pointcuts. It describes the graphical means as well as their semantics in terms of Object Constraint Language (OCL) [37] expressions. Section 4 presents some related work. Section 5 concludes the paper.

2. Pointcut Specification as a Distinct Design Issue

When looking at the specification of crosscutting features in aspect-oriented software development, we identify the specification of the elements (Figure 1, A) that crosscut a given decomposition and the specification of the set of join points (Figure 1, B) where that crosscutting takes place to be two fairly independent issues that can be seen as distinct design problems. We do this because we contemplate that the reasoning on the crosscutting assertions can be accomplished not knowing where exactly that crosscutting assertions are applied to; and vice versa, we suppose we can reflect on the join points at which crosscutting should occur while neglecting what exactly is to be inserted at these points.

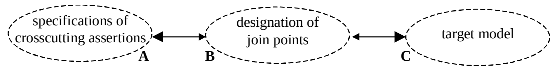
specifications of crosscutting assertions
A
B
designation of join points
C
target model

Figure 1. Aspect-oriented design issues (non-UML diagram) (cf. [33])

In an aspect-oriented software framework, for example, when we specify a particular synchronization strategy we do not want to determine yet which objects it should be applied to. Oppositely, we could identify the actions that need to be synchronized first, and leave it to a third party to design the synchronization strategy. In a third case, we have both a set of different synchronization strategies and a set of join point collections that designate actions to be synchronized. Now, we are able to combine crosscutting assertions and hooks of crosscutting in any manner, and thus we are set to realize any synchronization requirement by simple hooking the right strategy onto the right set of join points. In conclusion, the separate treatment of crosscutting details and points of crosscutting is a very important issue to achieve incremental programming (cf. [38]). And after all that's what we're heading for in aspect-oriented programming, too.

Looking at current aspect-oriented programming techniques we recognize that most of them commit to this separation of concern. AspectJ, for example, provides advice to specify crosscutting code, and pointcuts to specify where that code is to be introduced into the base program. We may combine advice and pointcuts in arbitrary manners: For example, we can specify an advice that hooks onto an "abstract pointcuts", i.e., empty sets of join points. That set can be filled by diverse subentities later-on for the crosscutting to take effect in multiple concerns. On the other hand, we can specify pointcuts first and let subentities implement the crosscutting behavior that is to be executed at those pointcuts. In Hyper/J [17], hyperslices designate all model elements in a given decomposition that belong to a particular concern (this process is call "concern mapping"). Two hyperslices may be composed by a hypermodule, which contains correspondence rules that determine at what points the hyperslices should be joined. Usually, we specify the hyperslices first and use a hypermodule to join one to the other afterwards. We could, however, also assign the composition of two concerns in advance and then define (or change) the hyperslices (or concern mappings) that are to be involved. That way we can substitute the crosscutting details to be introduced to a particular join point according to our need and desire. In Adaptive Programming [5], the affiliation between the specification of crosscutting details and the specification of crosscutting hooks is much stronger. However, even though we cannot change the one thing without adapting the other, Adaptive Programming distinguishes between traversal strategies that specify the locations at which crosscutting is to take place and visitor methods that specify how these locations are to be augmented.

Of course, even though we may reason on the specification of crosscutting details (Figure 1, A) and the specifications of the hooks (Figure 1, B) separately there certainly exist strong correlations between these two issues. In particular, dependencies arise from the charge of the latter to designate elements in the environment of the crosscut decomposition (Figure 1, C) that are used by the former. Nevertheless, we will neglect these kinds of dependencies for now and concentrate on the specifications of the hooks, i.e., on the designation of join points and of join conditions. For a closer elucidation on the dependencies between the specification of hooks and the specification of crosscutting details, please refer to [33].

3. Modeling Means for Pointcut Specification

On implementation level, join points represent "hooks where enhancements may be added"[1] (cf. [14]), e.g., classes or method calls. On modeling level, these join points are rendered by model elements in models – may it be a structural model describing a hierarchy of classifiers or a behavioral model describing control flow. On implementation level, pointcuts characterize the (sets of) hooks and/or (optional) conditions at/under which crosscutting takes place. For modeling pointcuts on modeling level, we thus need a means to render a set of model elements together with a set of conditions that must evaluate true for those model elements.

We choose UML Classifiers to represent join points in structural models, and UML Messages to represent join points in behavioral models. For the designation of join points – i.e., UML Classifiers and UML Messages, respectively – we introduce a new graphical means called "Join Point Designation Diagram" (JPDD). JPDDs resemble UML collaboration templates, however they lack the generative semantics of templates. That is, JPDDs describe "selection patterns" rather than "generation patterns". They specify all properties a model element (i.e., UML Classifier or UML Message) must provide in order to represent a join point (rather than the properties that will be added to or modified at those join points). These properties may be of structural or behavioral kind.

Structural properties are defined by means of class diagrams. Class diagrams may be used to model structural conditions of crosscutting, e.g., a particular feature or relationship that must be present for a classifier to represent a join point. Behavioral properties are defined by means of interaction diagrams (i.e., sequence diagrams or collaboration diagrams). Interaction diagrams are used, for example, to model behavioral conditions of crosscutting, e.g., that a particular message must be called within the control flow of some other message in order to represent a join point. JPDDs may contain both class diagrams and interaction diagrams in order to describe structural and behavioral properties at the same time (just like ordinary UML collaborations). Recall, though, that the semantics of class diagrams and interaction diagrams contained in JPDDs is different from their conventional variants as they specify a query on model elements rather than the model elements themselves.

The actual join points in a JPDD are modeled as JPDD's template parameters. JPDDs may designate both kinds of join points – i.e., UML Classifiers and UML Messages – at the same time.

The semantic of JPDDs is specified by means of OCL expressions: Each JPDD can be transformed into a OCL selection query picking out all model elements from a given UML model that represent join points. Those OCL statements make use of various meta-operations that we have appended to the UML meta-classes. Note that not all OCL operations are shown here due to limitations in space. Have a look at [6] to obtain the full OCL code.

In the following, we demonstrate with help of various examples what JPDDs look like and how they can be put to use to model various kinds of pointcuts. For each example, we present the relevant OCL expressions that are involved in matching UML Classifiers and UML Messages with the selection pattern described by a JPDD.

3.1. Pointcuts in the Early Aspects Stage

In the early aspect stage, JPDDs come in when we map aspect-oriented requirements to an aspect-oriented architecture. For example, Figure 2 shows two use cases that model two requirements, one (aspectual) synchronization requirement and some (core) functionality requirement which needs to be synchronized. The crosscutting relationship between one and the other has already been identified[2].

When mapping the requirements to an architecture, we must determine how the software artifact realizing the

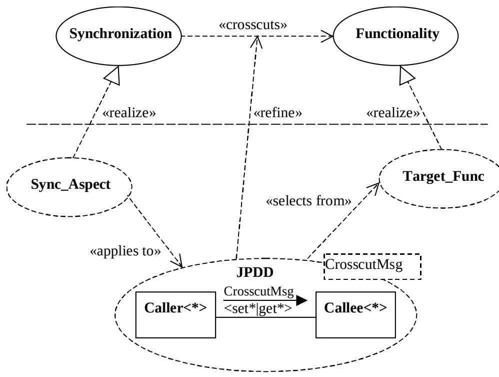

Figure 2. JPDDs in the early aspects stage

[1] Remember the difference we make here to join points in AspectJ terminology, where join points represent "principal points in the execution of a program" (cf. [10]).

[2] Note the subtle yet essential difference between «crosscut» relationships in aspect-oriented software development and «extend» relationships in use case driven software development (cf. [32]).

aspectual requirement connects to the software artifact realizing the functional requirement. This is the task of JPDDs. In Figure 2, for example, the two software artifacts are represented by collaborations. A JPDD is used to characterize the connection points at which the aspectual collaboration crosscuts the functional collaboration. In doing so, the JPDD gives some more details on the «crosscut» relationship between the use cases. The JPDD describes what is expected by the aspect, or what is exposed to the aspect, from the target environment so that it can accomplish its task. In a sense, the JPDD specifies an aspect-specific view on the target artifact.

We can learn from the example in Figure 2 that the synchronization aspect is concerned about synchronizing method calls ("CrosscutMsg") from one entity ("Caller") to another entity ("Callee"), which are expected to be some kinds of classifiers in the target artifact. Further, we can see that the aspect expects the methods' names to begin with "set" or "get". This name restriction may originate in a design decision on the aspectual or the functional requirement. For some (incomprehensible) reason, for example, we could be required to distinguish between synchronized "setter" and unsynchronized "putter" methods.

In the following we explain further capabilities of JPDDs to express views on the deployment environment of aspects and briefly sketch how they map to OCL expressions. For doing so, we chose to use examples from common aspect-oriented programming techniques to demonstrate the practical relevance of the designation means in JPDDs.

3.2. Pointcuts in AspectJ

Figure 3 models the following AspectJ pointcut (adopted from [21]):

```
pointcut aspectj_pc():
 cflowbelow(call(* ColoringClient.*(..))
    && this(SomeCaller))
 && call(FigureElement Figure.make*(..))
```

It designates all messages that invoke a method beginning with "make" on class "Figure" (returning an instance of class "FigureElement") from within the control flow of any method called on class "ColoringClient" from class "SomeCaller" (returning any or none return value). The message being crosscut is rendered as template parameter "CrosscutMsg".

Table 1 gives a general description on how message matching is accomplished – the depicted (meta-)operation "matchesMessage" is invoked on each message in the UML model. At first, the messages' names are matched. If the message in the JPDD is tagged with a

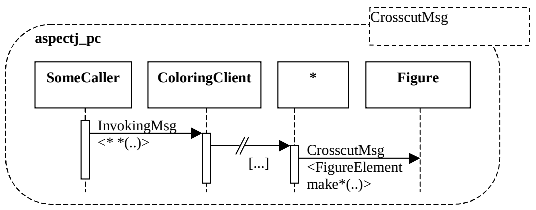

Figure 3. An AspectJ pointcut as JPDD

"joinPointPattern" (which are enclosed by sharp brackets "<...>"; see Figure 3), the "joinPointPattern"'s value (e.g., "FigureElement Figure.make*(..)") is passed for matching rather than the message's name (e.g., "CrosscutMsg"). Then, the message's sender and receiver are matched. This includes matching of their relationships (association roles, generalizations, etc.). After that, the associations used for transmitting the messages are compared.

Note that sender and receiver comparison is accomplished by matching the sender's and receiver's *role* in the JPDD to the sender's and receiver's *base* classifiers in the target model. This is because behavioral crosscutting takes place in every target model whose participants provide the set of features specified in the JPDD – may they be explicitly required by means of the role specification in the collaboration, or implicitly present by means of the base classifier specification in the class hierarchy. The same counts for the associations used for transmitting the messages.

In case the JPDD defines predecessors and/or an activator to the crosscut message (like "InvokingMessage" in Figure 3), the message in the target model must provide corresponding messages among its predecessors. The

Table 1. Matching messages in UML models

```
Context Message::
matchesMessage(m : Message) : Boolean
post: result =                  -- evaluate name pattern ('<...>')
if m.taggedValue->includes(tv | tv.type.name = 'joinPointPattern')
then
    self.matchesNamePattern(m.taggedValue->select(tv |
        tv.type.name = 'joinPointPattern').dataValue->at(1))
else
    self.matchesNamePattern(m.name)
endif
                                     -- evaluate sender/receiver/...
and self.sender.base->includes(C |
    C.matchesRelationships(m.sender) and
    C.matchesClassifier(m.sender))
and self.receiver.base->includes(C |
    C.matchesRelationships(m.receiver) and
    C.matchesClassifier(m.receiver))
and self.communicationConnection.base
    .matchesAssociation(m.communicationConnection)
                               -- evaluate predecessors/activator
and m.allPredecessors->union(m.activator)->reject(m2 |
    m2.stereotype->includes(st | st.name='indirect'))->forAll(m2 |
    self.allPredecessors->includes(M | M.matchesMessage(m2)))
                                               -- evaluate action
and self.action.matchesAction(m.action)
```

precise position is not important. For message matching, messages of stereotype "indirect" (denoted by double-striked-through lines; see Figure 3) are neglected. Their only purpose in JPDDs is to indicate auxiliary control flow the predecessors may provoke.

Finally, the message's actions are matched.

3.3. Traversal Strategies in Adaptive Programming

Figure 4 models the following traversal strategy in Adaptive Programming (adopted from [22], [23]):

```
*from* Conglomerat
 *bypassing* -> *,subsidiaries,*
 *via* Officer *to* Salary
```

The traversal strategy starts at object Conglomerat and traverses a characterized path to object Salary. It states that on its way through the class hierarchy the traversal must pass object Officer. At the same, it requires that traversal must not pass an association end named "subsidiaries".

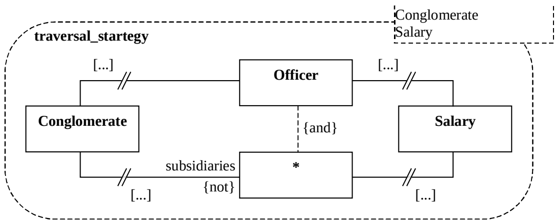

Figure 4. Traversal strategies as JPDD

In a UML model, the classifiers being traversed are identified with help of the (meta-)operation shown in Table 2. The operation analyzes if a classifiers possesses (a set of) associations matching to the ones specified in the JPDD. The operation distinguishes between standard associations and associations of stereotype "indirect"

Table 2. Matching associations in UML models

```
Context Classifier::
possessesMatchingAssociation(a : Association, c : Classifier) :
Boolean
post: result =                              -- evaluate indirect neighbours
if a.stereotype->includes(st | st.name='indirect') then
      self.associations->includes(A | A.matchesAssociation(a) and
      a.allConnections->select(ae | ae.participant = c)->forAll(ae |
         A.allConnections->select(AE | AE.participant = self)
            ->includes(AE | AE.matchesAssociationEnd(ae) and
           a.allConnections->select(ae | ae.participant <> c)
            ->forAll(ae2 | self.allIndirectNeighbors(A)
            ->includes(AE2 | AE2.matchesAssociationEnd(ae2))))))
else                                          -- evaluate direct neighbours
      self.associations->includes(A | A.matchesAssociation(a) and
      a.allConnections->forAll(ae | A.allConnections
            ->includes(AE | AE.matchesAssociationEnd(ae))))
endif
```

(denoted by double-striked-through lines; see Figure 4). In the former case, comparison is successful if the classifier provides a matching association with matching association ends. In the latter case, there must exist a navigable path from the current classifier to a classifier matching the associate in the JPDD. The association ends at which navigation starts and ends must match the association ends of the association specified in the JPDD.

For example, the bottom left association in Figure 4 denotes a navigation path starting with an association end whose participant is of type Conglomerate. And it ends with an association end whose name must not be "subsidiaries" – no matter of the type of its participant. From the participant, however, there must be a navigable path that ends with an association end whose participant is of type Salary (bottom right association in Figure 4).

3.4. Composition Rules on Declaratively Complete Hyperslices in Hyper/J

Composition rules in Hyper/J specify how the elements of one hyperslice are to be composed with the elements of another hyperslice. For that purpose, composition rules designate the join points in each hyperslice. Likewise, JPDDs are capable to designate model elements from UML models. While doing so, JPDDs may also reflect on the "declarative completeness" constraint in Hyper/J: In Hyper/J, each hyperslice needs to be "declaratively complete" (cf. [36]). That means that each hyperslice declares the structural properties it expects to be provided by another hyperslice. We can use JPDDs to model such structural requirements.

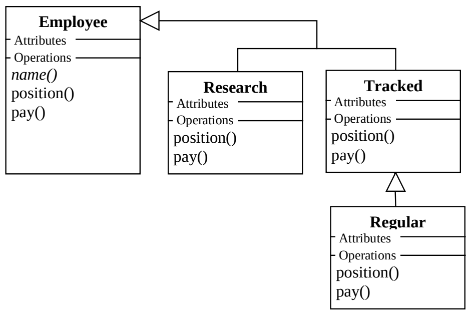

Figure 5. A payroll hyperslice (cf. [29])

For example, let's imagine a payroll hyperslice that implements "position()" and "pay()" operations on four classes "Employees", "Research", "Tracked", and "Regular" (see Figure 5). For their execution, the hyperslice requires the presence of a "name()" (declared as abstract in Figure 5), whose implementation must be provided by some other hyperslice – e.g., a personnel hyperslice (the example is adopted from [29]).

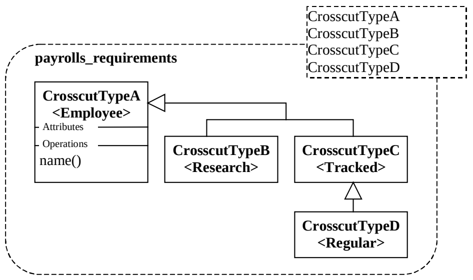

Figure 6. Specifying structural requirements in JPDDs

Figure 6 depicts a sample JPDD which designates the join points in the personnel hyperslice and specifies structural requirements being imposed on those join points. Table 3 and Table 4 describe the (meta-)operations for locating join points in UML models according to the specifications made in the JPDD.

Figure 6 depicts a JPDD that selects four classes from the personnel hyperslice as join points ("Employee", "Research", "Tracked", and "Regular"). These classes are meant to be augmented by the payroll hyperslice during composition. Besides designating the hyperslices' join points, though, the JPDD in Figure 6 specifies a couple of structural requirements that those join points must fulfill. At first, it requires class "Employee" to provide an operation "name()". Further, it requires class "Research" and "Tracked" to be subclasses of class "Employee", while class "Regular" in turn must be a subclass of class "Tracked". Composition may only take place if these constraints are satisfied.

Table 3. Matching classifiers in UML models

```
context Classifier::
matchesClassifier(C : Classifier) : Boolean
post: result =                              -- evaluate name pattern
if C.taggedValue->includes(tv | tv.type.name = 'joinPointPattern')
then
    self.matchesNamePattern(C.taggedValue->select(tv |
        tv.type.name = 'joinPointPattern').dataValue->at(1))
else
    self.matchesNamePattern(C.name)
endif
                                   -- evaluate defined meta-properties
and (self.isRoot = C.isRoot or C.isRoot = '')
and (self.isLeaf = C.isLeaf or C.isLeaf = '')
and (self.isAbstract = C.isAbstract or C.isAbstract = '')
                                  -- evaluate attributes and operations
and (C.feature->select(f | f.oclIsKindOf(Attribute))->forAll(ATT |
    self.possessesMatchingAttribute(ATT))
    or C.feature->select(f | f.oclIsKindOf(Attribute))->size = 0)
and (C.feature->select(f | f.oclIsKindOf(Operation))->forAll(OP |
    self.possessesMatchingOperation(OP))
    or C.feature->select(f | f.oclIsKindOf(Operation))->size = 0)
```

Table 3 describes how join points are selected from UML models. Join point selection is accomplished by name matching. If the classifier specifications in the JPDD are tagged with a "joinPointPattern" (which are enclosed by sharp brackets "<...>"; see Figure 6), the "joinPointPattern"'s value (i.e., "Employee", "Research", "Tracked", or "Regular") is passed for matching rather than the classifiers' names (e.g., "CrosscutTypeA", etc.). Apart from their names, the classifier's meta-properties must match ("isRoot", "isLeaf", "isAbstract"). At last, the classifiers' features, i.e., attributes and methods, are compared. A classifier must possess all attributes and all methods being defined as structural requirement in the JPDD (like operation "name()" in Figure 6, for example) in order to be selected as join point.

Further, classifiers must possess all relationships, i.e., associations, generalizations, and specializations, that are defined in the JPDD (like the inheritance relationships in Figure 6, for example) in order to be selected as join point. Matching of relationships is accomplished by a second (meta-)operation for associations, generalizations, and specializations separately (see Table 4).

Table 4. Matching relationships in UML models

```
context Classifier::
matchesRelationships(B : Classifier) : Boolean
post: result =                              -- evaluate relationships
 (B.parent->forAll(P |
    self.possessesMatchingParent(P))        or B.parent->size = 0)
and (B.child->forAll(CH |
    self.possessesMatchingChild(CH))        or B.child->size = 0)
and (B.associations->forAll(A |
    self.possessesMatchingAssociation(A, self))
                                          or B.associations->size = 0)
```

4. Related and Future Work

A couple of other approaches deal with modeling pointcuts using OCL, UML, and even MDA:

[31] makes use of OCL code [27] to bind elements form an application models to "hot spots" in aspect-oriented frameworks. In doing so, they select model elements that are to be enhanced like we do. Unlike us, however, they define the enhancements in the same OCL statement which hinders reuse of the query specification. Specifying enhancements is not duty of JPDDs.

A more sophisticated approach is described in [15] which presents a domain-specific extension to the OCL for the specification of crosscutting constraints. In particular, it introduces reflective operators to advance selection of model elements. Again, though, selection queries and modification assignments are instantly coupled together, and so, reuse of queries is not possible.

[34] [16] chooses to use UML Action Semantics [28] to define model transformations and OCL [37] to express selection criterions for those transformations. As queries are hard-coded into transformations, they cannot be reused in a different context.

[24] discusses how Model-Driven Architecture (MDA) [25] may support aspect-oriented modeling. It points out that a pointcut can be expressed as a query on one model. We share that conception and have defined a graphical notation to define such queries. We see another application area of our approach in connection with the Query View Transformation Language (QVT) [26] which is currently under review by the OMG: JPDDs can be used as a graphical query language to select model elements from UML models that are subject to transformations.

Besides that we see complementary contribution of our work to existing aspect-oriented modeling and design approaches, for example [13], [32], and [19], which lack graphical means to specify selection queries. Moreover, as JPDDs map onto OCL expressions, our approach can be seamlessly integrated into [16] and [15]. From using parameterized OCL (meta-)operations, we even gain greater flexibility because we may feed the operations with different JPDDs at a time.

Future work will involve investigations on how to specify selection queries in the context of aspect-oriented modeling with state charts [8] or activity diagrams [11]. Further, JPDDs are to be integrated into a UML profile for aspect-oriented modeling (cf. [7]) in order to advance its application in the aspect-oriented software development process.

5. Conclusion – Going Beyond

In this paper we have exemplified the need for distinct modeling means for the specifications of pointcuts, i.e., the specification of places and conditions at/under which crosscutting takes place. We presented a graphical notation that suits this purpose, and we have exemplified its use and semantics when designing a synchronization requirement in an aspect-oriented manner as well as with help of examples from different aspect-oriented implementation techniques.

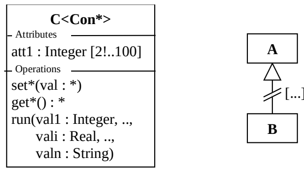

Figure 7. Going beyond in aspect-oriented modeling

Yet, note that the capabilities of our modeling notation go beyond the designation means of current aspect-oriented implementation techniques and allow advanced aspect-oriented modeling. The stereotype “indirect”, for example, is not limited to association relationships (denoted by double-striked-through lines; see Figure 4) but to generalization and specialization relationships as well (denoted by double-striked-through lines with hollow arrow heads; see Figure 7 right side). Using this symbol in JPDDs signifies that a given classifier must provide an ancestor or a descendant, respectively, that matches the specification of the JPDD.

Besides that the notation provides for the specification of operations using wildcards “*” and “..” in their parameter list (see Figure 7 left side for an example). Further, we allow the specification of multiplicity ranges for attributes (see Figure 7 left side). Classifiers representing join points must provide a matching attribute whose multiplicity resides in the range specified by the JPDD (e.g., “[2..100]”). An exclamation mark denotes a fixed lower or upper bound (e.g., “2!”). Please refer to [6] for the corresponding OCL code.

Provided with these novel modeling means, software designers are capable to design pointcuts in wholly new ways. Being implementation language independent, the modeling notation allows design of pointcuts in the very early stages of software development, e.g., when designing connection points in aspect-oriented software architectures. Further, aspect-oriented software developers may fully concentrate on design first, and finally can map their design to whatever aspect-oriented programming language seems best suited.

6. References

[1] 1st Workshop on Aspect-Oriented Modeling, at AOSD'02 (Enschede, The Netherlands, Apr. 2002), http://lgl.epfl.ch/workshops/aosd-uml/index.html

[2] 2nd Workshop on Aspect-Oriented Modeling, at UML’02 (Dresden, Germany, Sep. 2002), http://lglwww.epfl.ch/workshops/uml2002/

[3] 3rd Workshop on Aspect-Oriented Modeling, at AOSD'03 (Boston, MA, Mar. 2003), http://lglwww.epfl.ch/workshops/aosd2003/

[4] 4th Workshop on Aspect-Oriented Modeling, at UML’03 (San Francisco, CA, Oct. 2003), http://www.csam.iit.edu/~oaldawud/AOM/

[5] Adaptive Programming, http://www.ccs.neu.edu/research/demeter/

[6] Addendum to Stein, D., Hanenberg, S., Unland, R., Aspect-Oriented Modeling in the Light of MDA, submitted to the Special Issue of Science of Computer Programming (Elsevier) on Model Driven Architecture: Foundations and Applications, http://dawis.informatik.uni-essen.de/site/staff/stein/

[7] Aldawud, O., Bader, A., Elrad, T., *UML Profile for Aspect-Oriented Software Development*, 3rd AOM Workshop at AOSD'03 (Boston, MA, Mar. 2003)

[8] Aldawud, O., Bader, A., Elrad, T., *Weaving with Statecharts*, 1st AOM Workshop at AOSD'02 (Enschede, The Netherlands, Apr. 2002)

[9] AspectJ, http://www.aspectj.org

[10] AspectJ Team, *The AspectJ Programming Guide*, http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/aspectj-home/doc/progguide/index.html, Jan. 2004

[11] Barros, J.P., Gomes, L., *Towards the Support for Crosscutting Concerns in Activity Diagrams: A Graphical Approach*, 4th AOM Workshop at UML'03, (San Francisco, CA, Oct. 2003)

[12] Booch, G., Jacobson, I., Rumbaugh, J., *The Unified Modeling Language User Guide*, Addison Wesley, Reading, MA, 1999

[13] Clarke, S., Walker, R.J. *Composition Patterns: An Approach to Designing Reusable Aspects*. in Proc. of ICSE '01 (Toronto, Canada, May 2001), ACM, 5-14

[14] Elrad, T., Aksit, M., Kiczales, G., Lieberherr, K., Ossher, H., *Discussing Aspects of Aspect-Oriented Programming*, in: ACM Communications, Vol. 44(10), Oct. 2001, pp. 33-38

[15] Gray, J., Bapty, T., Neema, S., Schmidt, D.C., Gokhale, A., Natarajan, B., *An Approach for Supporting Aspect-Oriented Domain Modeling*, in: Proc. of GPCE '03 (Erfurt, Germany, Sep. 2003), Springer, pp. 151-170

[16] Ho, W.M., Jézéquel, J.-M., Pennaneac'h, F., Plouzeau, N., *A Toolkit for Weaving Aspect Oriented UML Designs*, in: Proc. of AOSD '02 (Enschede, The Netherlands, Apr. 2002), ACM, pp. 99-105

[17] Hyper/J, http://www.alphaworks.ibm.com/tech/hyperj

[18] Jacobson, I., Booch, G., Rumbaugh, J., *The Unified Software Development Process*, Addison Wesley, Reading, MA, 1999

[19] Kandé, M.M., PhD Thesis, EPFL, Lausanne, Swiss, 2003

[20] Katara, M., Katz, Sh., *Architectural Views of Aspects*, in: Proc. of AOSD'03 (Boston, MA, Mar. 2003), ACM, pp. 1-10

[21] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G., *Getting Started with AspectJ*, ACM Communications, Vol. 44(10), Oct. 2001, pp. 59-65

[22] Lieberherr, K., *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*, PWS Publishing Company, Boston, 1996

[23] Lieberherr, K., Orleans, D., Ovlinger, J., *Aspect-Oriented Programming with Adaptive Methods*, ACM Communications, Vol. 44(10), Oct. 2001, pp. 39-41

[24] Mellor, St., *On A Framework for Aspect-Oriented Modeling*, 4th AOM Workshop at UML'03, (San Francisco, CA, Oct. 2003)

[25] Object Management Group (OMG), *MDA Guide Version 1.0*, May 2003

[26] Object Management Group (OMG), *Request for Proposal: MOF 2.0 Query / Views / Transformations RFP*, Apr. 2002

[27] OMG, *Response to the UML 2.0 OCL RFP*, Revised Submission, Version 1.6, January 2003

[28] OMG, *Unified Modeling Language Specification*. Version 1.5, Mar. 2003

[29] Ossher, H., Tarr, P., *Using Multi-Dimensional Separation of Concerns to (Re)Shape evolving Software*, in: ACM Communications, Vol. 44(10), Oct. 2001, pp. 43-50

[30] Rashid, A., Moreira, A., Araújo, J., *Modularisation and Composition of Aspectual Requirements*, in: Proc. of AOSD'03 (Boston, MA, Mar. 2003), ACM, pp. 11-20

[31] Rausch, A., Rumpe, B., Hoogendoorn, L., *Aspect-Oriented Framework Modeling*, 4th AOM Workshop at UML'03, (San Francisco, CA, Oct. 2003)

[32] Stein, D., Hanenberg, St., Unland, R., *A UML-based Aspect-Oriented Design Notation For AspectJ*, in: Proc. of AOSD '02 (Enschede, The Netherlands, Apr. 2002), ACM, pp. 106-112

[33] Stein, D., Hanenberg, St., Unland, R., *Issues on Representing Crosscutting Features*, 3rd AOM Workshop at AOSD'03 (Boston, MA, Mar. 2003)

[34] Sunyé, G., Pennaneac'h, F., Ho, W.-M., Le Guennec, A., Jézéquel, J.-M., *Using UML Action Semantics for Executable Modeling and Beyond*, in: Proc. of CAiSE'01 (Interlaken, Switzerland, Jun. 2001), Springer, pp.433-447

[35] Sutton, St., Rouvellou, I., *Modeling of Software Concerns in Cosmos*, in: Proc. of AOSD '02 (Enschede, The Netherlands, Apr. 2002), ACM, pp. 127-133

[36] Tarr, P., Ossher, H., *Hyper/J User and Installation Manual*, IBM Corp., 2000

[37] Warmer, J., Kleppe, A., *The Object Constraint Language: Precise Modelling with UML*, Addison-Wesley, 1998

[38] Wegner, P., Zdonik, S., *Inheritance as Incremental Modification Mechanism or What Like is and Isn't Like*, in: Proc. of ECOOP'88 (Oslo, Norway, Aug. 1988), pp. 55-77