On Representing Join Points in the UML

Dominik Stein, Stefan Hanenberg, and Rainer Unland

Institute for Computer Science

University of Essen, Germany

{dstein | shanenbe | unlandR}@cs.uni-essen.de

ABSTRACT

Join points represent the key concept in Aspect-Orientation. Join points define the places where two concerns crosscut one another. It is a major task for aspect-oriented designers to specify a set of join points at which two concern models are (inter)connected to each other. Hence, it is a primary task for an aspect-oriented modeling language to provide suitable representations for join points. In our Aspect-Oriented Design Model we have identified join point representations in the UML that serve the needs of aspect-oriented designers of aspect-oriented programs written in AspectJ. In this paper we evaluate if and to what extent these representations are apt to serve as hooks for crosscutting specified with Composition Filters, in Adaptive Programming, and in Hyper/J, as well. Based on the outcome of that investigation we present a graphical notation for the designation of join points and for their visualization in regular user models.

1. INTRODUCTION

Join points represent the key concept in aspect-orientation: Join points describe the "hooks" where crosscutting enhancements may be added to a given decomposition (cf. [7]). Join points identify the points of correspondence in independently modularized concerns. Join points define the places where two concerns crosscut one another. The nature of a join point importantly influences the way in which crosscutting at that point can be accomplished. For example, the different join point models used by the prevalent aspect-oriented programming techniques like AspectJ [3], Composition Filters [1], Adaptive Programming [12], and Hyper/J [9] open up different possibilities to implement crosscutting entities. Furthermore, the different characteristics (i.e., the different temper) of join points specified in their join point models require different means of join point designation. Hence, the actual disposition of join points lies at the heart of an aspect-oriented environment.

Due to the significance of join points in an aspect-oriented environment it is a major task for aspect-oriented designers to specify (sets of) join points at which two concern models are (inter)connected to each other. Correspondingly, it is a primary task for an aspect-oriented modeling language to provide suitable representations for join points – both on meta(model)-level and on (user)model-level. On model level, the need for appropriate representations is most obvious since it is the principal purpose of a modeling language to provide designers with graphical means to visualize software systems. Yet, to allow integrity and consistency checking on the design of those software systems, representations on the meta-level are needed as well.

With the Aspect-Oriented Design Model (AODM) [15] [16] we present such a modeling language that provides join point representations both on meta-level and on model-level. The AODM is based on the UML [13] and uses its standard extension mechanisms to allow the design of structural and behavioral crosscutting. On the meta-level, the AODM identifies UML classifiers to represent join points for structural crosscutting and UML links to represent join points for behavioral crosscutting. UML classifiers constitute supertypes of entities like classes, interfaces, nodes, or components. UML links are runtime instances of UML associations and represent communication channels that are used to pass control via message sending from one object to another. On the model-level, the AODM renders (sets of) join points using textual expressions that evaluate to (sets of) references to classifiers and links, respectively. The references are visualized in user models by means of special «crosscut» relationships, which are newly introduced by the AODM.

The AODM has been developed as a modeling language for the design of aspect-oriented programs realized with AspectJ. As a result, the AODM primarily focuses on the appropriate representation of AspectJ's language constructs. And the representation of join points in the AODM primarily reflects on AspectJ's join point model. Therefore, investigations are due to find out how the AODM can be extended to provide for other aspect-oriented programming techniques.

In this paper we present our preliminary results on the extension of the AODM to other aspect-oriented programming techniques with respect to the representation of join points. We investigate if and to what extent UML classifiers and UML links may serve as (meta-level) join points for aspect-oriented crosscutting defined with Composition Filters, Adaptive Programming, and Hyper/J (section 2). Besides that we explore how join points can be graphically rendered on model-level (section 3). At last we summarize the outcomes of this work and give a short outlook on the remaining adaptations to the AODM that need to be accomplished (section 4).

2. JOIN POINTS ON META-LEVEL

In this section we explain what we understand by structural and behavioral crosscutting and why we consider UML classifiers and UML links to depict appropriate representations for join points for structural and behavioral crosscutting. We demonstrate why UML classifiers and UML links may serve as meta-level representations for join points for aspect-oriented crosscutting defined with AspectJ, Composition Filters, Adaptive Programming, and Hyper/J.

2.1 Structural vs. Behavioral Crosscutting

In the following examinations we distinguish between "structural crosscutting" and "behavioral crosscutting". We do this in close conformity with AspectJ. That means, with

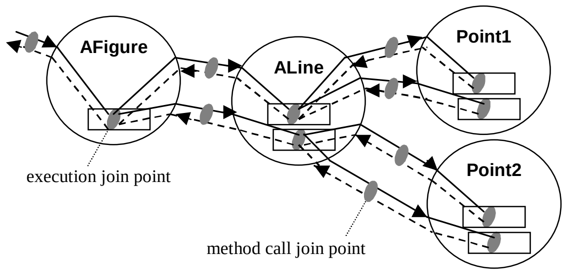


Figure 1. Simple Runtime Object Call Graph (cf. [8])

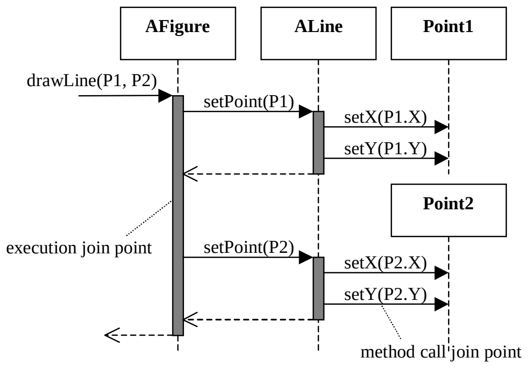


Figure 2. UML Sequence Interaction Diagram

"structural crosscutting" we refer to crosscutting that affects the type structure of a given model, e.g., the interfaces of classifiers, or their relationships. We identify structural crosscutting to apply to some spatial location (where?) in a class hierarchy. Therefore, we take UML classifiers as hooks for structural crosscutting. In contrast to this we use the term "behavioral crosscutting" to refer to crosscutting that affects the (existing) behavior of a classifier. Behavioral crosscutting applies to some point in time (i.e., at runtime) (when?). We take UML links as hooks for behavioral crosscutting because they demarcate fixed points that are repeatedly passed during runtime. Note that in the case of structural crosscutting the element serving as hook for crosscutting coincides with the element being affected by crosscutting (i.e., it is a UML classifier). Behavioral crosscutting, in contrast to this, hooks on to links; however, it does not affect the link. Rather, it affects the (inter)action being performed over that link.

Hence, for example, we consider a method being added to a classifier by a crosscutting concern to be structural crosscutting because the addition of a method affects the interface of the classifier rather than its (existing) behavior. That is, the method constitutes a new service, however, it does not crosscut an existing one.

2.2 Join Points in AspectJ

Aspect-Oriented Programming [11] with *AspectJ* [3] is used to specify both structural and behavioral crosscutting. Structural crosscutting is specified by means of "introductions" that change "the type structure of a program, by adding to or extending interfaces and classes with new fields, constructors, or methods" [2]. Hence, structural crosscutting in AspectJ applies to interfaces and classes. In the UML meta-model, interfaces and classes are modeled as subtypes of UML classifiers. Therefore, we think it is proper to hook on structural crosscutting defined in AspectJ to classifiers in UML models. And thus, we consider it eligible to use UML classifiers to represent AspectJ' s join points for structural crosscutting in the UML.

Behavioral crosscutting in AspectJ is specified by special kinds of methods (i.e., pieces of "advice") that execute at "well-defined point in the program flow" [2]. These well-defined points "can be considered as nodes in a simple runtime object call graph" [10]. In the UML, these nodes are depicted as UML links in UML interaction diagrams. For example, Figure 1 and Figure 2 demonstrate how an object call graph (taken from the AspectJ Tutorial [8]) can be transformed into a UML interaction diagram. In the interaction diagram, links denote the "well-defined points" at which behavioral crosscutting is performed. The diagram demonstrates how behavioral crosscutting defined in AspectJ programs hooks on to links in UML models. Therefore, we consider it eligible to use UML links to represent AspectJ' s join points for behavioral crosscutting in the UML.

2.3 Join Points in Composition Filters

Composition Filters [1] are used to specify behavioral crosscutting (in the sense specified in subsection 2.1). That is because Composition Filters "are defined as functions, which manipulate messages received and sent by objects" [4]. In the UML, messages are depicted in interaction diagrams. And as mentioned before, links are used to communicate these

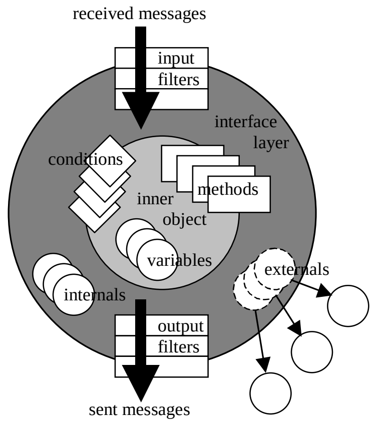


Figure 3. The Composition Filters Object Model (cf. [5])

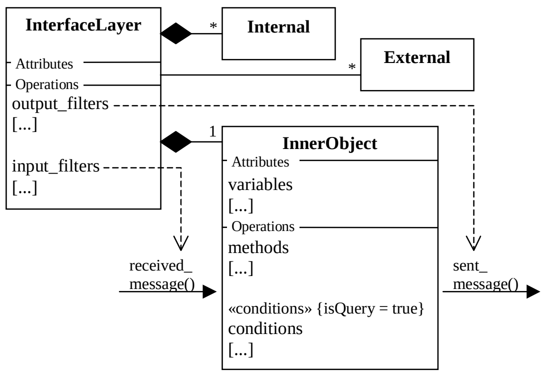


Figure 4. UML Collaboration Interaction Diagram

messages from one object to another. In general, no message can be sent from one object to another in UML models, if there is no link connecting these objects. So, just like behavioral crosscutting in Aspect-Oriented Programming with AspectJ, behavioral crosscutting defined with Composition Filters hooks on to links in UML models. And correspondingly, we consider UML links eligible to represent join points for behavioral crosscutting defined by Composition Filters.

Figure 3 and Figure 4 demonstrate how the Composition Filters Object Model [5] can be represented by an UML interaction diagram. Note in Figure 4 how the input and output filter methods are hooked on to the link transmitting the received and sent messages (respectively).

2.4 Join Points in Adaptive Programming

Adaptive Programming [12] with "adaptive methods" is used to implement collaborative behavior between interrelated classes without hard-coding their relationships. Adaptive methods hook on to join points that are thought of "as nodes or edges in some graph" [7]. The join points are succinctly specified by means of a general traversal strategy, which can be applied to a particular user graph such as "a dynamic call graph (a UML interaction diagram), a class graph (a UML class diagram), an object graph (a UML object diagram)" [7]. Each node (i.e., each join point) in such a graph is supplemented with a so-called "visitor method". These visitor methods represent helper methods, which collaboratively realize the behavior specified by the adaptive method.

While we clearly recognize that the main objective of an adaptive method in Adaptive Programming is to specify collaborative behavior of interconnected classes, we identify crosscutting in Adaptive Programming to be a kind of structural crosscutting (in the sense stated in subsection 2.1). We do this because we observe that the propagation of the visitor methods along the traversal strategies in fact alters interfaces of classifiers rather than (existing) behavior (or services) of those classifiers. This means for a UML model, visitor methods hook on to classifiers. Therefore, we consider UML classifiers eligible to represent hooks for structural crosscutting defined in Adaptive Programming.

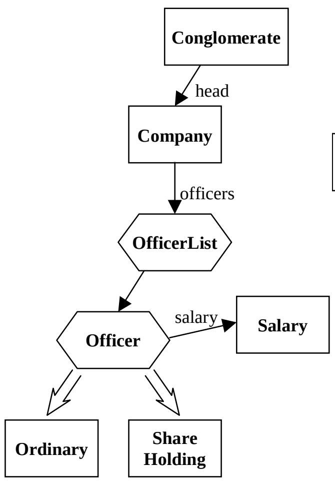


Figure 5. Traversal Strategy Propagation Graph (cf. [12])

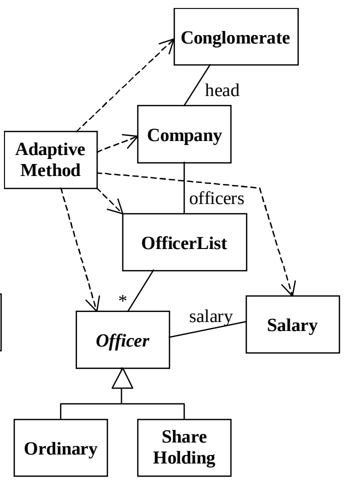


Figure 6. UML Class Diagram

For example, Figure 5 and Figure 6 demonstrate what a possible propagation graph for a given traversal strategy (taken from [12]) could look like in a UML model and how the propagation of an adaptive method along this graph can be represented.

2.5 Join Points in Hyper/J

At last we examine to what extent UML classifiers and UML links in a UML model may serve as hooks for crosscutting implemented with *Hyper/J* [9]. To do so, we regard a special case of model composition with Hyper/J, which compares to the model composition of other aspect-oriented programming techniques. Afterwards, we contemplate the general case.

Generally, Hyper/J can be used to implement structural and behavioral crosscutting just as in other aspect-oriented programming techniques. In [14] Ossher and Tarr point out, though, that there is "a key difference between MDSOC with Hyper/J and AOP as described in the literature [11] and exemplified by AspectJ [10]" in "that AspectJ supports augmentation of a single model, whereas Hyper/J supports integration of multiple models." Note in that regard, that we observe Composition Filters and Adaptive Programming, as well, to support the augmentation of a single model (likewise to AspectJ) rather than an integration of multiple models (as promoted by Hyper/J). To help us understand the parallels of Hyper/J to the other aspect-oriented programming techniques, we imitate the "augmentation" process of the latter in the former and think of a composition of two models as an insertion of one ("aspect") model into the other ("base") model (i.e., we use merge integration strategies and assume that no conflicts need to be reconciled and no elements need to be renamed or retyped).

In Hyper/J, crosscutting hooks on join points that "include classes, interfaces, methods, and member variables" [7]. These join points delineate points of correspondence between two (or more) separate models. They are related by means of composition relationships, which specify further details on the exact composition. We identify crosscutting in Hyper/J (with one exception described later) to be a kind of structural crosscutting (in the sense defined in subsection 2.1) because we recognize that Hyper/J' s primarily concern is to integrate type hierarchies (cf. "Hyper/J allows a developer to compose a collection of separate models, [...] each [...] implementing a (partial) class hierarchy"[1] [14]). We observe that composition rules connect to (spatial) locations in type hierarchies (rather than instants in the execution of a program). That is, composition rules render overlapping parts in type hierarchies. Figure 7, for example, depicts two independent concern models (taken from [14]) that are related by composition relationships (not all are shown) that designate corresponding elements in both models.

In our scenario described above we may abstract from composition rules that connect member attributes or member operations not implemented in the "aspect" model. We observe that these composition relationships exist solely due

[1] full citation: "Hyper/J allows a developer to compose a collection of separate models, called hyperslices, each encapsulating a concern by defining and implementing a (partial) class hierarchy appropriate for that concern." [14]

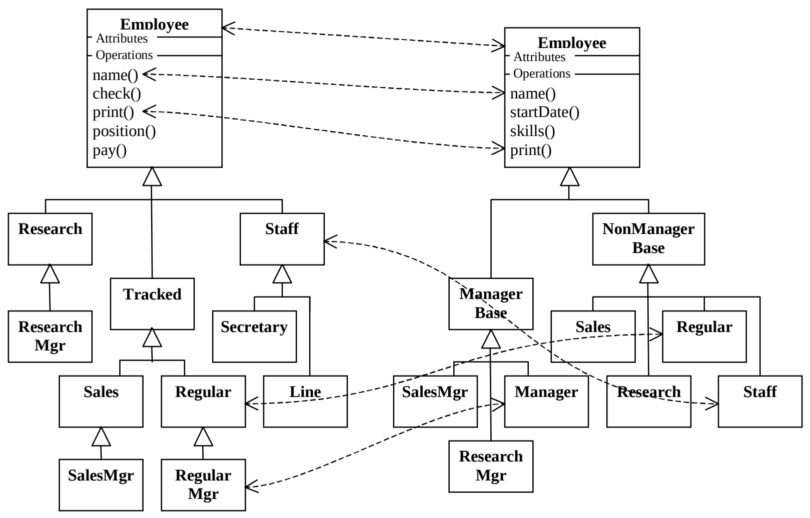


Figure 7. Composition Relationships between Two Distinct Concern Models (cf. [14]) represented as UML Class Diagrams

to Hyper/J's "declarative completeness" requirement [17], which means that each type hierarchy must declare every member attribute and every member operation that it refers to. We recognize that none of these attributes and operations will be injected into the "base" model during composition – because they are already present. Only those attributes and operations that are not attached to a composition relationship (and thus do not represent join points) will be inserted. So, in our scenario described above we can abstract from "method" join points and "member variable" join points – ending up with "class" and "interface" join points as the major hooks for structural crosscutting in Hyper/J. As mentioned before, in the UML meta-model, classes and interfaces represent subtypes of UML classifiers. So, structural crosscutting on classes and interfaces defined in Hyper/J hooks on to classifiers in UML models. Therefore, we think UML classifiers represent suitable join points for structural crosscutting defined in Hyper/J.

Special regards must be given to composition rules that relate member operations implemented in both the "aspect" and the "base" model. Such composition rules express behavioral crosscutting in Hyper/J (in the sense stated in subsection 2.1). Behavioral crosscutting in Hyper/J is specified on method level, i.e., methods are considered atomic. Therefore, specifying behavioral crosscutting in Hyper/J principally means specifying the order in which corresponding methods are to be invoked. In the UML, method invocations are represented as call messages, which are communicated over links. So, just like behavioral crosscutting in Aspect-Oriented Programming or with Composition Filters, behavioral crosscutting defined in Hyper/J hooks on to links in UML models. Therefore, we consider UML links eligible to represent join points for behavioral crosscutting defined in Hyper/J.

In conclusion, interpreting the composition of two models in Hyper/J as an insertion of one ("aspect") model into the other ("base") model helped us to recognize that UML classifiers and UML links in a UML model represent eligible hooks for crosscutting implemented with Hyper/J. However, these hooks do not prove to be sufficient to express all possible integration strategies of Hyper/J in UML models. We also need to represent method and attribute join points for the general case where we cannot abstract from method and attribute correspondence relationships.

We relate the reason of the need for such representations to the fact that in Hyper/J developers are concerned with the composition process of "declaratively complete" models itself, while in other aspect-oriented programming techniques weaving is a pre-defined process executed on known models. Hyper/J programmers need to declare everything to which they refer, while programmers using other aspect-oriented programming techniques reference directly to the used elements. For that reason the former need to explicitly designate corresponding elements, while the latter rely on the implicit correspondence relationships implemented by their compilers and weavers.

Clarke [6] presents a powerful approach that focuses on the specification of such explicit correspondence (and detailed composition) relationships on the design level. The approach provides representational means to designate attribute and operation join points. Our AODM, on the contrary, realizes implicit correspondence as it is found in AspectJ by means of parameterization (using template parameter or operation parameters, respectively; see [15] [16] for further details). It does not provide for the design of relationships between corresponding attribute and operation join points.

3. JOIN POINTS ON MODEL-LEVEL

As an important outcome of the last section we identified UML classifiers and UML links to be appropriate meta-level representations for join points in AspectJ, Composition Filters, Adaptive Programming, and Hyper/J. However, identifying join points on the meta-level is only half of the story. Designers must also be provided with graphical means to identify join points on model-level. These means ought to serve the design of any aspect-oriented programming technique. That means in particular, we need modeling facilities to graphically represent pointcuts in AspectJ, messages for Composition Filters, traversal strategies in Adaptive Programming, and composition relationships in Hyper/J. In our AODM, we use a (specifically introduced) «crosscut» relationship to point to the locations where a crosscutting element of an "aspect" model crosscuts an element of a "base" model. Figure 4 and Figure 6 give a basic idea of how this relationship looks like.

Using a relationship to indicate the locations where an "aspect" element crosscuts a "base" element (i.e., to express crosscutting between an "aspect" element and a "base" element) appears to be quite befitting at the first glance. At the second glance, however, the approach proves to be suitable only for the visualization of «crosscut» relationships between a small number of "aspect" and "base" elements. As soon as we need to display *all* crosscutting relationships in a design model, the model is likely to become cluttered and sprawling. To overcome this tangle we suggest the use of separate modeling means for the *designation* of points of crosscutting

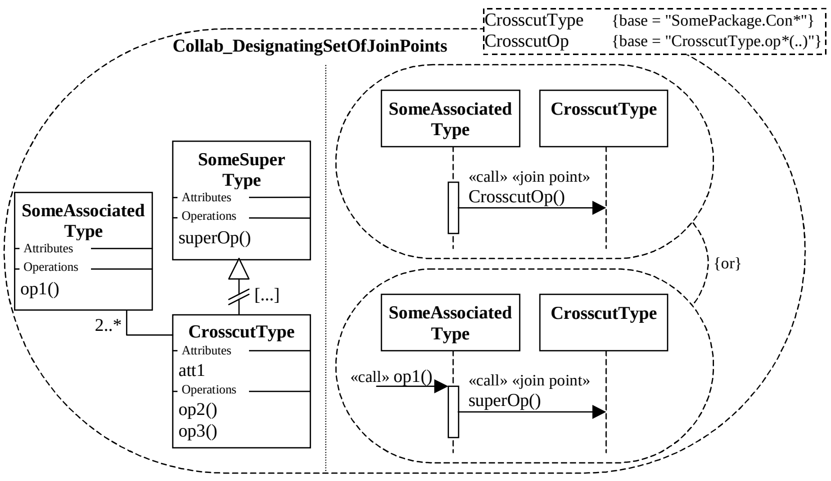
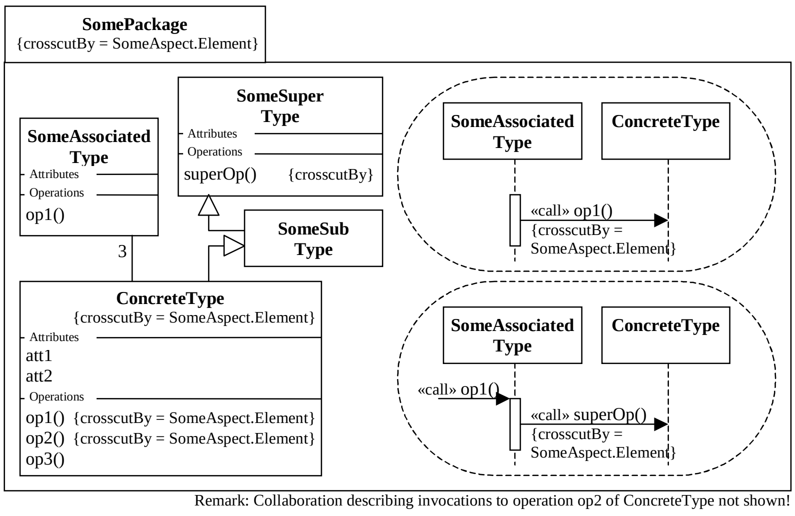


Figure 8. Join Point Designation Diagram



SomePackage
{crosscutBy = SomeAspect.Element}
SomeAssociated Type
Attributes
Operations
op1()
SomeSuper Type
Attributes
Operations
superOp() {crosscutBy}
SomeSub Type
3
ConcreteType
{crosscutBy = SomeAspect.Element}
Attributes
att1
att2
Operations
op1() {crosscutBy = SomeAspect.Element}
op2() {crosscutBy = SomeAspect.Element}
op3()
SomeAssociated Type
ConcreteType
«call» op1()
{crosscutBy = SomeAspect.Element}
SomeAssociated Type
ConcreteType
«call» op1()
«call» superOp()
{crosscutBy = SomeAspect.Element}
Remark: Collaboration describing invocations to operation op2 of ConcreteType not shown!

Figure 9. Join Point Indication Diagram

in "aspect" models on the one hand and for the *indication* of those points in "base" models on the other hand. In the following we present solutions for the separate designation and indication of join points in user models.

3.1 Designating Join Points

For the graphical designation of join points, we propose to use special "join point designation diagrams" that basically represent UML collaborations. In these diagrams, join points (i.e., classifiers in the case of structural crosscutting and links in the case of behavioral crosscutting; cf. section 2) are indicated as special «join point» stereotypes.

Figure 8 demonstrates, for example, how a set of join points for behavioral crosscutting (i.e., links) is specified. The left part of the collaboration outlines some requirements on the structural environment of links belonging to the designated set of join points. The right part shows two interactions that designate the messages that must be transmitted over those links. Each interaction designates a different (i.e., alternative) link at which crosscutting is to take place (denoted by the `{or}` relationship).

The upper interaction designates links that connect instances of `SomeAssociated-Type` to instances of `CrosscutType` and that are used to transmit a call message to invoke a `CrosscutOp` operation on a `CrosscutType` instance to be a join point. The `CrosscutType` instances and the `CrosscutOp` operations are designated by means of textual pattern expressions specified in special `base` tags. The `base` tags are specified in a template parameter box to avoid distractions inside the collaboration. The lower interaction designates links that connect instances of `SomeAssociatedType` to instances of `CrosscutType` and that are used to transmit a call message to invoke the (inherited) `superOp` operation on a `CrosscutType` instance. The call message needs to be transmitted in the execution context of an `op1` operation invoked on a `SomeAssociatedType` instance.

In the left part, the collaboration declares structural requirements that must be met by a link to be appended to the set of join points. For instance, `CrosscutType`s are required to be subtypes of `SomeSuperType` and to have at least two associations to instances of `SomeAssociatedType` (the disconnected generalization relationship between `CrosscutType` and `SomeSuperType` indicates that the former does not necessarily have to be a direct child of the latter). The supertype `SomeSuperType` is required to provide a `superOp` operation, which is inherited to the `CrosscutType`s. Instances of `SomeAssociatedType` are required to provide an `op1` operation. The `CrosscutType`s themselves are demanded to feature an attribute `att1` and two operations `op2` and `op3`.

Join points for structural crosscutting can be specified by collaboration diagrams, too. In that case, the collaboration only shows structural requirements and does not contain interaction diagrams. Join points are designated in the structural description by marking classifiers with the «join point» stereotype.

3.2 Indicating Crosscut Elements

After we found suitable representational means for the designation of join points we now look for lucid means to indicate join points in "base" models. To do so, we propose the use of special "join point indication diagrams" that describe a particular view (or projection) on a given user model. The view comprises only those model elements being affected by a particular set of join points (defined by a "join point designation diagram").

Figure 9, for example, demonstrates what a "join point indication diagram" for the set of join points defined by the "join point designation diagram" depicted in Figure 8 could look like. The diagram displays all elements pertaining to the particular set of join points together with their features and relationships. Each "base" model element is supplemented with a special `crosscutBy` tag, which is supposed to enumerate all "aspect" elements that crosscut that "base" element. For container "base" elements (e.g., the class `ConcreteType`) the `crosscutBy` tag lists all "aspect" elements that crosscut the "base" elements contained in the container. `CrosscutBy` tags may be shown without the detailed enumeration to simply mark a "base" element as being crosscut (e.g., the operation `superOp` of `SomeSuperType`).

Note that appropriate tool support is needed to keep "join point indication diagrams" consistent with their corresponding "join point designation diagrams".

4. SUMMARY AND FUTURE WORK

In this paper we have shown that UML classifiers and UML links are appropriate to represent join points in AspectJ, Composition Filters, Adaptive Programming, and Hyper/J on the meta-level. We recognized that Hyper/J calls for further means to represent the explicit correspondence relationships between operation and attribute join points. We identified relationships to be unsuitable to represent the (entire) crosscutting relationships between "aspect" and "base" model elements. We suggested separating the designation of join points in "aspect" models from their indication in "base" models. We have presented a special graphical notion of "join point designation diagrams" to designate sets of join points. We have proposed to mark model elements being affected by a particular set of join points with help of UML tags in special "join point indication diagrams".

Provided with (meta-level and model-level) representations for join points that appropriately delineate the join points used in the prevalent aspect-oriented programming languages (i.e., AspectJ, Composition Filters, Adaptive Programming, and Hyper/J), we can now move on to devise suitable modeling facilities for the design of crosscutting structure and crosscutting behavior, itself. In our AODM, we use (template and ordinary) collaborations to specify structural and behavioral crosscutting as it is realized with AspectJ. Its abstractions primarily suit the needs of aspect-oriented programmers working with AspectJ. Further investigations are required to explore if and to what extent these modeling facilities can be (re)used to design aspect-oriented programs with Composition Filters, Adaptive Programming, or Hyper/J. The preliminary investigations presented in this paper help to do that job. The extension to Composition Filters and Adaptive Programming promises to bring forth quick results since these approaches – likewise to AspectJ – are concerned with augmentation. The extension to Hyper/J is expected to raise more problems and therefore will be left aside for subsequent research.

5. REFERENCES

[1] Aksit, M., Bergmans, L., Vural, S., *An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach*, in: Proc. of ECOOP'92 (Utrecht, The Netherlands, Jun. 1992), LNCS 615, pp. 372-395

[2] AspectJ Team, *The AspectJ Programming Guide*, http://aspectj.org/doc/dist/progguide/index.html, Sep. 2001

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

[4] Bergmans, L., Aksit, M., *Composing Crosscutting Concerns using Composition Filters*, in: ACM Communications, Vol. 44(10), Oct. 2001, pp. 51-57

[5] Bergmans, L., *The Composition Filters Object Model*, Dept. of Computer Science, University of Twente, 1994

[6] Clarke, S., *Composition of Object-Oriented Software Design Models*, PhD Thesis, Dublin City University, Dublin, Ireland, Jan. 2001

[7] 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

[8] Griswold, W.G., Hilsdale, E., Hugunin, J., Ivanovic, V., Kersten, M., Kiczales, G., Palm, J., *Tutorial: Aspect-Oriented Programming with AspectJ,* Xerox Corp., 2001

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

[10] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, K., Palm, J., Griswold, W.G., *An Overview of AspectJ*, in: Proc. of ECOOP '01 (Budapest, Hungary, Jun. 2001), LNCS 2072, pp. 327-252

[11] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, Ch., Lopes, Ch., Loingtier, J.-M., Irwin, J., *Aspect-Oriented Programming*, in: Proc. of ECOOP '97 (Jyväskylä, Finland, Jun. 1997), LNCS 1241, pp. 220-242

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

[13] Object Management Group (OMG), *Unified Modeling Language Specification*, Version 1.4, Sep. 2001

[14] 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

[15] 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

[16] Stein, D., Hanenberg, St., Unland, R., *Designing Aspect-Oriented Crosscutting in UML*, AOSD-UML Workshop at AOSD '02 (Enschede, The Netherlands, Apr. 2002)

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