Query Models

Dominik Stein, Stefan Hanenberg, and Rainer Unland

University of Duisburg-Essen
Essen, Germany
{dstein, shanenbe, unlandR}@cs.uni-essen.de

Abstract. The need for querying software artifacts is a new emerging design issue in modern software development. Novel techniques such as Model-Driven Architecture or Aspect-Oriented Software Development heavily depend on powerful designation means to allocate elements in software artifacts, which are then either modified by transformation or enhanced by weaving processes. In this paper we present a new modeling notation for representing queries using the UML. We introduce special symbols for common selection purposes and specify their OCL selection semantics, which may be executed on existing UML models in order to allocate all selected model elements therein. By doing so, we aim to give forth the advantages of modeling to query design: Our query models facilitate the specification of queries independent from particular programming languages, ease their comprehension, and support their validation in a modeling context.

1 Introduction

Querying software artifacts is a new emerging design issue in modern software development. Novel software development techniques, such as Model-Driven Architecture (MDA) [9] and Aspect-Oriented Software Development (AOSD) [6], focus on the allocation of elements, which are then either modified by transformation (in MDA) or enhanced by weaving processes (in AOSD). The primary goal of these approaches is to apply common refinements to multiple points in target artifacts. At the same time, refinements are kept separate from the points being affected in order to allow reuse of refinements in different application domains.

Prerequisite to querying software artifacts is the existence of accurate query specification means since only accurate specification of the selection criteria on target elements may lead to desired results and avoid unpredicted effects. Accordingly, the need for query specification means in MDA is manifested in the "MOF 2.0 Query / Views / Transformation (QVT)" Request For Proposal (RFP) [10]. The RFP calls for suggestions for a standard model transformation language. One part of the RFP demands appropriate designation means to allocate model elements in existing models that will serve as sources to transformations. Most submissions to the RFP (e.g., [4], [12], [1]) propose a textual language – in particular, the Object Constraint Language (OCL) [17] – in order to query existing user models. However, using a textual language (like OCL) quickly leads to very complex expressions even when defining a relatively small number of selection criteria. Hence, as query comprehension is difficult, accurate query specification is not easy and error-prone. We feel that a

T. Baar et al. (Eds.): UML 2004, LNCS 3273, pp. 98-112, 2004.
© Springer-Verlag Berlin Heidelberg 2004

graphical notation could help. However, no graphical notation is currently around that assists the specification of selection queries in a more feasible manner.

In AOSD, the need of query specification means emerges from the need to specify sets of points in the target program (so-called "join points" [6]) at which aspect-oriented refinement shall take place. In order to designate such sets of join points, aspect-oriented programming languages provide special constructs, called "crosscuts" [3] (examples are "pointcuts" in AspectJ [7] and "traversal strategies" in Adaptive Programming [8]). Crosscuts are usually expressed using textual means. As with any textual pattern description, crosscuts as well tend to be difficult to comprehend as soon as they become more complex. Again we think that a graphical notation could help in understanding and specifying crosscuts. However, although there are various modeling approaches around to model aspect-oriented software (e.g., [2], [14], [5]), they all lack a suitable notation to represent selections of join points graphically.

In conclusion, we think that there is a need for a general graphical representation to express selection queries on software artifacts for the various domains. Indeed, we believe that such a suitable modeling notation is indispensable for technologies like MDA and AOSD to become popular software developing techniques. Such a graphical visualization would facilitate the comprehension of selection queries, as well as the estimation of where refinements actually take place. It would assist the software developer to communicate his/her ideas to colleagues or to document design decisions for maintainers and administrators. At last, provided with precise selection semantic, the modeling notation would permit reasoning on design decisions and validation of final results.

In this paper, we present a novel modeling notation for specifying selection queries: "Join Point Designation Diagrams" ("JPDD"). JPDDs are special kinds of diagrams that are used to visualize the selection criteria that elements must satisfy in order to be selected by the query. The notation is based on the modeling means provided by the Unified Modeling Language (UML) [11], making its comprehension easy and intuitive to a broad range of developers. The modeling notation is accompanied by a set of OCL operations that outline the allocation of elements in existing models according to the specifications made in a JPDD. These OCL operations allow for the validation of selection queries in a modeling context.

The remainder of this paper is structured as follows: At first, we emphasize the urgency of having a common graphical notation to represent queries giving real-life examples from the MDA and AOSD domains. After that, we introduce the selection means that we have defined and specify their selection semantics with the help of OCL expressions. We further describe the general syntax of JPDDs in order to elucidate the ways that our selection means may be combined. Finally, we demonstrate the applicability of our diagrams by applying them to the examples given in the problem statement. We conclude the paper with a short summary.

2 Problem Statement

In order to motivate the need to visualize query specifications, we present three daily-life sample implementations of queries as we can find them in MDA and AOSD.

2.1 Query Specification in OCL

The first example shows an (excerpt from an) OCL query statement as it may be used in MDA transformations. The statement selects all classes which are named "cn" and that either have an attribute named "an" or – in case not – have an association to some other class named "cn1" which in turn has an attribute named "an". The example is adopted from [12]. There it is used within a transformation that translates classes into tables. Fig. 1 shows a possible application area for such a transformation: Imagine we want to store all instances of "Person" in a database table together with the "Street" the person lives on. In some cases (A), the developers may have decided to implement "Street" as a direct attribute of "Person". In other cases (B), they may have chosen to associate class "Person" with another class "Address" which in turn includes the attribute "Street". The query we specify here covers both solutions.

```
->select(c: Class |
    (c.name='cn' and
     c.allAttributes->exists(att | att.name='an') )
or  (c.name='cn' and not
     c.allAttributes->exists(att | att.name='an') and
     c.oppositeAssociationEnds->exists(ae |
        let c1 : Class = ae.participant in
        c1.name='cn1' and
        c1.allAttributes->exists(att | att.name='an') ) ) )
```

The example demonstrates the complexity that the textual notation of OCL imposes on the developer when specifying or comprehending a selection query: He/she needs to be have a profound knowledge on what properties (e.g., `name` or `participant`) of what elements (e.g., classes and associations) may be constrained, and how. Further, the precise way to refer to the element's relationships to other model elements must be well understood (e.g., by using the collection operation `->exists` on properties like `allAttributes` or `oppositeAssociationEnds`). The assignment of variables (e.g., `c1`) using the `let` expression and the scoping of variable names by means of brackets is another source of significant complexity. Last but not least, the placement of Boolean operators is crucial to the selection result, and therefore must be carefully investigated.

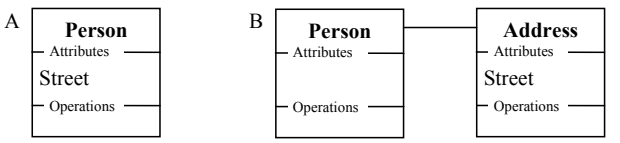

Fig. 1. Application areas for the OCL query

2.2 Query Specification in AspectJ

The next example shows a "pointcut" as we can find it in AspectJ [7]. AspectJ is a very popular general-purpose aspect-oriented programming language based on Java. Its "pointcut" construct is used to designate a selection of join points. A specialty of AspectJ is to allow aspect-oriented refinements (i.e., crosscutting) based on the runtime context of join points.

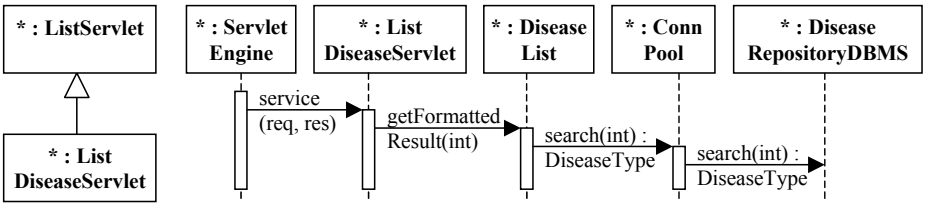

Fig. 2. Application area for the AspectJ pointcut

The sample pointcut shown here selects all those messages as join point that invoke (`call`) method "search" on class[1] "DiseaseRepositoryDBMS" (taking an integer value as parameter and returning an instance of class "DiseaseType") that come to pass within the control flow (`cflow`) of any (`*`) method called from (`this`) class[1] "ServletEngine" on class[1] "ListServlet" (or any of its subclasses[1] (`+`)), taking any number (`..`) of parameters and returning any or none (`*`) return value. The pointcut is adopted from an example in [13] and its main purpose is to reduce loading time of complex data objects ("DiseaseType") when only partial information is needed. Fig. 2 shows a possible scenario for our sample pointcut. In this scenario, class[1] "ServletEngine" invokes method "service" on class[1] "ListDiseaseServlet", which is a subclass of class[1] "ListServlet". After two hops via "DiseaseList" and "ConnPool", the message sequence ends invoking method "search" on class[1] "DiseaseRepositoryDBMS". At this point the selection criteria specified in the pointcut are satisfied and message "search" is added to the list of selected join points.

```
pointcut aspectj_pc():
    call(DiseaseType DiseaseRepositoryDBMS.search(int)) &&
    cflow(call(* ListServlet+.*(..)) && this(ServletEngine))
```

This example points out the absolute need of being familiar with keywords (such as `call` and `cflow`) and operators (like `+`, `*`, and `..`) for the comprehension of selection queries. In response to this indispensable prerequisite, we carefully related our explanations in the previous paragraph to the terms and characters used in the pointcut. However, in order to compose a new query, knowing the meaning of keywords and operators is not enough. Developers must be aware of what arguments may be specified within a particular statement (such as `call` and `cflow`), and what consequences these have. They need to know, for example, that the selection of `calls` can be further refined to operations having a particular signature pattern, or being invoked by certain instances (using the `this` construct). In the end, it requires high analytical skills in order to combine these statements (by means of Boolean operators) in such a way that only those join points are selected that we actually want to crosscut.

2.3 Query Specification in Demeter/C++

Our last example is from the domain of Adaptive Programming [8]. The goal of Adaptive Programming is to implement behavior in a "structure-shy" way. That means that methods should presume as little as possible about the (class) structure

[1] i.e., an instance of that class/those classes

they are executed in. For that purpose, Adaptive Programming makes use of special kinds of crosscuts, so-called "traversal strategies".

The following traversal strategy is taken from [8]. It selects a path from class "Conglomerate" to class "Salary" that passes class "Officer", however, that does not include an association (end) of name "subsidiaries". The strategy is part of a method that sums up the salaries of "Officers" in the "Conglomerate". Fig. 3 shows a possible class hierarchy that the method can cope with. The actual summation is accomplished by visitor methods that are appended to the individual classes on the strategy path. Note that according to the traversal strategy, calculation does not consider officers employed by the company's subsidiaries.

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

Looking at this example, we are once again faced with new keywords and new operators whose meaning and consequences must be well understood by the developer. A major part of complexity arises from the various kinds of relationships – denoted as construction, alternation, inheritance, and repetition edges – that can be specified in traversal strategies: Developers need to be aware of the distinctive meaning of each of those edge types, and they must remember their precise notation (`->`, `=>`, `:>`, and `~>`, respectively) in order not to designate the wrong relationship. At last, they have to keep in mind what specific information they may provide with each relationship. For example, restrictions to the relationship labels may only be specified for construction edges.

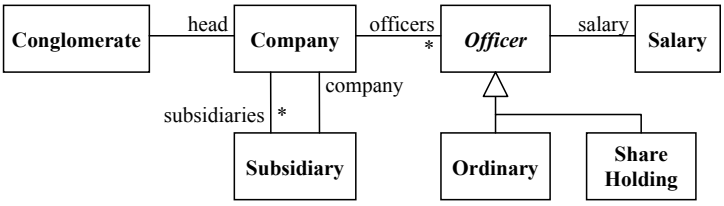

Fig. 3. Application area for the Demeter/C++ traversal strategy

2.4 Preliminary Conclusion

As a conclusion from the previous investigations, we attest textual notations to leave developers stranded with a heavy load of complexity: Developers are required to have a profound understanding about the usage of keywords and operators in order to specify queries properly. They must know what properties of what elements they may refer to, and how. Finally, they must have high analytical capabilities in order to assess the grouping of selection criteria as well as their semantic interdependencies, so that they can estimate what elements will be actually retrieved.

Opposed to so much complexity, developers urgently call for a graphical notation that help them with the composition and comprehension of selection queries. That notation should give them a visual conception of the selection semantics they are currently specifying using textual keywords. It should facilitate the specification of

selection criteria on elements and their properties. Furthermore, it should depict the grouping of such selection criteria and visualize their interdependencies.

In order to come up with more concrete characteristics that our graphical notation must possess, we revisit the examples of the previous sections and investigate what different kinds of selections we have used: First of all, we may observe that – even though each notation comes with its own, most individual syntax, keywords, and operators – they are all concerned with the selection of (more or less) the same program elements, namely classes and objects, as well as the relationships between them (i.e., association relationships, generalization relationships, and call dependencies). Further, we recognize that selection is almost always based on the element names (only sometimes the elements name does not matter). Apart from that, selection may be based on the element's structural composition; for example, based on the (non-)existence of features in classes or of parameters in a parameter list. The last observation we make is that selection of elements is often based on the general context they reside in; meaning that query specifications abstract from (a set of) direct relationships between elements and merely call for the existence of paths.

Having identified these core objectives of a graphical query language, we now explain how we deal with these issues in our query models.

3 Modeling Selection Criteria

In this section we present the core modeling means we developed for the specification of selection criteria in selection queries. We explain their graphical notation, describe their objectives, and define their selection semantics using OCL meta-operations. Due to space limitations, only important meta-operations are shown.

Before discussing the modeling means in detail, we like to emphasize some general facts: Each model element is selected based on the values of their meta-attributes. In doing so, we extrapolate our observation that selections may be based on the value of the element's meta-attribute "name", and allow selections based on the values of the other meta-attributes, as well. Further, selection may be based on the model element's meta-relationships to other elements. That way we cope with the occasions when elements need to be selected based on their structural composition. Evaluation of meta-attributes and meta-relationships is accomplished by special OCL meta-operations, which we append to various meta-classes in the UML meta-model (see Table 1 for an example). These OCL meta-operations take a selection criterion from a JPDD as argument and compare it with an actual model element in a user model.

Within the meta-operations, name matching is generally accomplished using name patterns. Name patterns may contain wildcards, such as "*" and "?", in order to allow the selection of groups of elements (of the same type) following similar naming conventions. Within a JPDD, each model element's name is considered to be a name pattern by default. A name pattern may be given an identifier in order to reference the name pattern's occurrences at another place within the JPDD. Graphically, such identifiers are prepended by a question mark and are enclosed by angle brackets. They are placed in front of the name pattern whose occurrences they reference (see "<?C>Con*" in Fig. 4 for an example). Technically, name patterns which are given an identifier are stored in a special tagged value, named "namePattern". Is such a tagged value present, the OCL meta-operation evaluates the tagged value for name matching rather than the model element's proper name (see Table 1, block I, for details).

A JPDD must always show all characteristics that are considered relevant for selection. If any meta-attribute is not explicitly set to a value, or if any meta-relationship is not explicitly defined to be present, they are regarded irrelevant for selection (like the publicity specification in Fig. 4, for example). In doing so, we allow selections based only on partial information. Special treatment is necessary whenever selection criteria are defined on values of meta-attributes that are mapped to standard representations in diagrams. For example, every class in a UML class diagram is non-abstract by default – unless explicitly stated otherwise. Without additional means, it would not be possible to select classifiers regardless of the value of those meta-properties. To overcome this dilemma, the values of meta-attributes may always be explicitly defined using standard constraint notation (see Fig. 4 for an example).

3.1 Classifier Selection

To demonstrate the general facts mentioned above, let us have a look at the way that classifier selections may be specified. Fig. 4 depicts the graphical means that we provide for defining selection criteria on classifiers. Table 1 details how such means are evaluated on existing UML models using OCL expressions. As you can see, the OCL meta-operation first evaluates the model element's name (or the tagged value "namePattern", if present). Then, it compares the element's meta-attributes. And finally, it considers the element's meta-relationships to other model elements.

When specifying selection criteria on classifiers, special regards must be given to the features they must or must not possess. As illustrated in Fig. 4 (see attribute "att1"), we can use the Boolean operator "{not}" in order to require the non-existence of a particular element for selection. Technically, the matching result is inverted by the OCL meta-operation in that case (see [16] for further details).

Further, you can choose the multiplicity of attributes to indicate exact upper and/or lower limits or to designate upper and/or lower bounds which the multiplicity of an attribute must not exceed or underrun (respectively). The lower multiplicity limit of "att2" in Fig. 4, for example, is an exact lower limit (indicated by "!"). Attributes are only selected, if their lower limit equates "2". The upper multiplicity limit of "att2" in

Table 1. OCL meta-operation for matching classifiers

```
context Classifier::
matchesClassifier(C : Classifier) : Boolean
post: result =                                          -- block I. evaluate name pattern
if C.taggedValue->exists(tv | tv.type.name = 'namePattern') then
   self.matchesNamePattern(C.taggedValue->select(tv |
      tv.type.name = 'namePattern').dataValue->asSequence()->at(1))
else
   self.matchesNamePattern(C.name)
endif                                                   -- block II. evaluate meta-attributes
and (self.isRoot = C.isRoot or C.isRoot = '')
and (self.isLeaf = C.isLeaf or C.isLeaf = '')
and (self.isAbstract = C.isAbstract or C.isAbstract = '')
                                                        -- block III. evaluate meta-relationships
and (C.allAttributes->forAll(ATT | self.possessesMatchingAttribute(ATT))
   or C.allAttributes->size() = 0)
and (C.allOperations->forAll(OP | self.possessesMatchingOperation(OP))
   or C.allOperations->size() = 0)
```

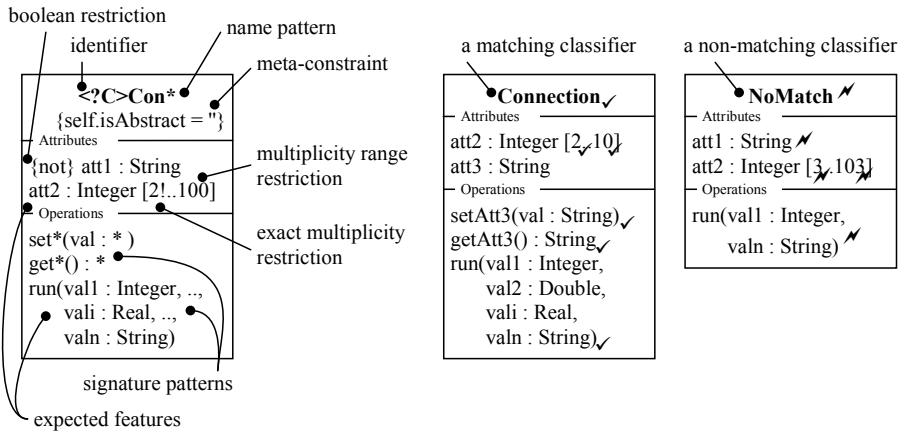

Fig. 4. Classifier selection

Fig. 4 denotes an upper bound. Attributes are selected if their upper multiplicity limit does not exceed "100" (see Fig. 4, right part, for examples). Technically, fix upper and lower limits are indicated by special stereotypes: Attributes of stereotype "fixed-LowerLimit" determine a fixed lower limit. Attributes of stereotype "fixedUpper-Limit" determine a fixed upper limit (see [16] for further details). Graphically, attributes of both stereotypes are indicated by appending an "!" to the respective multiplicity limit (see Fig. 4 for example).

Finally, operations are specified using signature patterns, which may contain wildcard ".." in order to abstract from an arbitrary number of parameters in the operation's parameter list (see Fig. 4 for an example). Matching is accomplished by comparing the overall order of the parameters in the operation's parameter list, as well as their particular order at the beginning and the end of the parameter list (see [16] for further details on the OCL matching expressions).

3.2 Relationship Selection

As pointed out in section 2, the presence of relationships and in particular the existence of paths between elements plays an important role in selection queries. In the following we present the graphical means we provide to deal with both association, generalization, and specialization relationships and paths, respectively.

Association Selection. Fig. 5 depicts the graphical means we provide for specifying selections based on association relationships. Table 2 details how such means may be evaluated on existing UML models using OCL expressions. Note that, similar to features, we may restrict the multiplicity of association ends and/or require an association (end) *not* to be present (see Fig. 5, center part, for an example). Special regards must be given to "indirect" associations, or association paths. Graphically, indirect associations are depicted as double-crossed lines. Fig. 5, left part, for example, signifies that there must be an association path from class "C" to class "AC". Technically, indirect associations are indicated using a special stereotype "indirect" (see Table 2 for details).

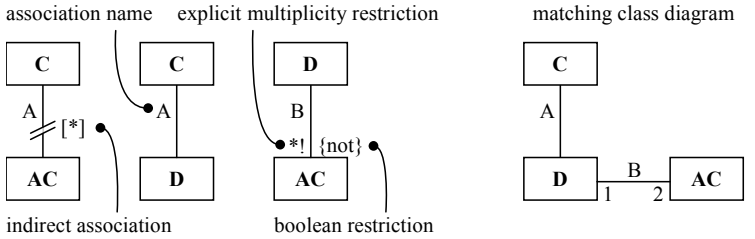

Fig. 5. Association selection

Matching of paths essentially means comparing the ends of the path in a UML model with the ends of the indirect association in the JPDD. Table 2 details how this is accomplished using OCL expressions: Operation "indirectNeighbors" returns all navigable association ends of a given association. Operation "allIndirectNeighbors" returns all navigable association ends that are reachable via a given association. Operation "possessesMatchingAssociation" then makes use of this operation to evaluate whether one of such association ends matches the opposite association end of the indirect association (see block I).

Generalization and Specialization Selection. Fig. 6 depicts the graphical means we provide for defining selection criteria on generalization and specialization relationships. Table 3 details how such means may be evaluated on existing UML models using OCL expressions. Note that, as with association relationships, we may

Table 2. OCL meta-operation for matching association relationships

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

```
context Classifier::
allIndirectNeighbors(a : Association): Set(AssociationEnd)
post: result = self.indirectNeighbors(a)->union(
   self.indirectNeighbors(a)
   ->collect(AE | AE.participant.allAssociations->reject(A | A = a)
   ->collect(A | AE.participant.allIndirectNeighbors(A))))->asSet()
```

```
context Classifier::
indirectNeighbors(a : Association): Set(AssociationEnd)
post: result = self.allOppositeAssociationEnds
   ->select(AE | AE.association = a and AE.isNavigable)
```

specify a generalization/specialization relationships *not* to be present (see Fig. 6, center part, for an example).

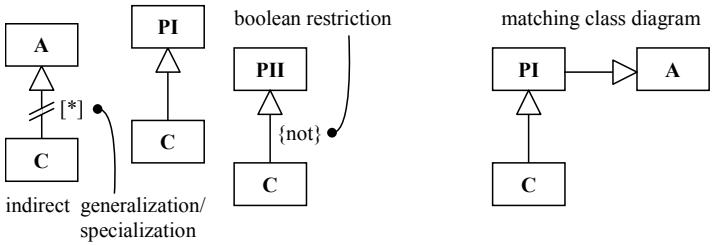

Fig. 6. Generalization selection

Special regards must be given to "indirect" generalization and specialization relationships, or inheritance paths (see Fig. 6). Graphically, indirect generalizations and specializations are depicted as double-crossed lines. According to the specification made in Fig. 6, left part, for example, class "C" must have class "A" among its ancestors in order to be selected, and class "A" must have class "C" among its descendants. Technically, inheritance paths are represented as special stereotype "indirect" of the generalization relationship. Table 3 exemplifies how inheritance path matching is accomplished in UML models in case of an generalization relationship (see block I).

Table 3. OCL meta-operation for matching generalization relationships

```
context Classifier::
possessesMatchingParent(g : Generalization) : Boolean
post: result =                                    -- block I. evaluate indirect parents
if g.stereotype->exists(st | st.name='indirect') then
   self.generalization->exists(G | G.matchesGeneralization(g) and
   G.parent->union(G.parent.allParents)->exists(C |
       C.matchesClassifier(g.parent) and
       C.matchesRelationships(g.parent)))
else                                              -- block II. evaluate direct parents
   self.generalization->exists(G | G.matchesGeneralization(g) and
       G.parent.matchesClassifier(g.parent) and
       G.parent.matchesRelationships(g.parent))
endif
```

3.3 Message Selection

As demonstrated in section 2, selections are not confined to the structural properties of software artifacts, but may be based on their behavior, as well. In the following, we describe the graphical means we provide to specify such selections. In doing so, we concentrate on the symbols used in UML sequence diagrams (i.e., messages).

Messages are selected based on the actions they are associated with. Signature patterns may be used to restrict such actions (see Fig. 7, left part, for example). Besides that, messages may be selected based on the control flow they occur in or which they invoke. Such control flow is sketched using predecessor and successor messages, respectively: All messages complying to message "?msg1" in Fig. 7, center

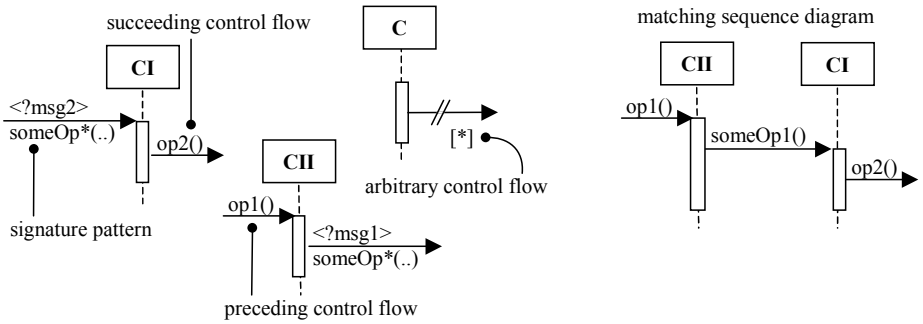

Fig. 7. Message selection

part, for example, must occur in the control flow of message "op1". All messages selected by message "?msg2" in Fig. 7, left part, must invoke some message "op2". Special regards must be given to "indirect" messages, which may be used to indicate an arbitrary control flow. Graphically, indirect messages are depicted as double-crossed arrows (see Fig. 7, center part, for example). Technically, they are represented as messages of the special stereotype "indirect". Last but not least, it is important to note that messages are selected based on the base classifiers of their sender and receiver roles (rather than on the roles themselves). This is accomplished deeming that selections should execute on the full specification of classifiers rather than on restricted projections. The same is valid for the associations used for transmitting the messages. See [16] for the precise OCL code used for message matching.

4 A Technical Perspective

Having specified the graphical notation and their semantics in the previous section, we now want to describe briefly how those means are integrated into the UML. Fig. 8 depicts the general syntax of a JPDD: It consists of at least one selection criterion, some of which delineate selection parameters. A JPDD represents a selection criterion itself, and thus may be contained in another JPDD (e.g., for reuse of criteria specifications). Fig. 8 illustrates how we can map the syntax of JPDDs to the general syntax of UML namespace templates: In terms of the UML meta-model, a JPDD represents a (special stereotype of a) namespace which may contain several model elements, each representing a selection criterion. The namespace is provided with a set of template parameters that indicates the model elements to be returned by the query. We do not further restrict the particular kind of namespace that a JPDD may reify since different application domains may have different demands. Therefore, JPDDs may be specified as classifier templates, collaboration templates, or package templates, etc. – whatever suits the needs of the particular query specification best.

It is important to note that JPDDs do not quite comply with the semantic of conventional UML templates. In fact, the meaning of JPDDs is rather "inverse" to that of conventional UML templates: While conventional UML templates are generally used to instantiate multiple model elements from one common mould (or a "generation pattern"), JPDDs are used to identify all model elements that share one common shape (a "selection pattern"). Correspondingly, template parameters of

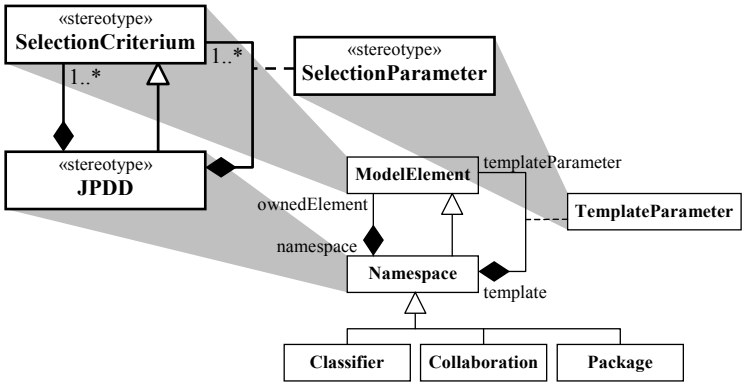

Fig. 8. Abstract syntax of JPDDs, and its mapping to UML's meta-classes

JPDDs are meant to return actual arguments rather than being bound to actual arguments. To indicate this important difference in meaning visually, we place template parameter boxes to the bottom right corner of JPDDs (rather than to their conventional position at the top right corner of the template). In the following section we give examples of what a fully specified JPDD looks like.

5 Application to Software Development Techniques

In this section we demonstrate how our notation may be put to use in actual software development techniques, namely in MDA and AOSD. To do so, we revisit the examples given in section 2 and show how these sample query specifications can be represented using JPDDs. We demonstrate the benefits that our modeling notation yields to the comprehension of query specifications and compare it to other possible approaches to visualize selection queries.

5.1 Model-Driven Architecture

It has been already mentioned that the need to specify model queries in the field of MDA is manifested in OMG's "MOF 2.0 QVT" RFP. While most submissions to the RFP are content with proposing textual notations (in particular OCL) to specify model queries, only one [12] comes up with a graphical representation. In the following we compare that graphical notation with the one presented here. To do so, we make use of the OCL selection statement described in section 2.1.

Let us first have a look at the JPDD (see Fig. 9, left part). The JPDD depicts three classes (together with their features) and one relationship. The elements are grouped into two alternative selection patterns, which are interconnected by a Boolean "{or}". Attribute "?att" in the upper class of the right selection pattern is annotated with a Boolean "{not}", stating that no matching attribute must be present in the respective classifier for the selection to succeed. Class names and attributes names are specified using name patterns ("cn", "cn1", and "an"), which are given identifiers ("?c", "?c1", and "?att"). Two of those identifiers ("?c" and "?att") reappear in the template

parameter box of the query, meaning that the selection is supposed to return all class/attribute-pairs that satisfy the specified selection criteria in the JPDD.

Fig. 9, right part, shows a graphical representation of the same query using the notation presented in [12]. The approach aims to define general meta-model mappings in MDA. Therefore, queries are defined in terms of meta-model entities and meta-model properties rather than in terms of user model entities and user model properties (as in our approach). In consequence, the approach can be considered to be in parts more general than ours: Its notation allows the specification of model queries for any (MOF) meta-model and is not confined to the UML meta-model. However, we think that the gain of higher generality is at cost of lower feasibility and readability: Users have to learn and understand the meta-models they are (unawarely) working with before they can write their model queries. Further, the need to express selection criteria in terms of meta-model entities may often lead to unnecessary and distracting noise in selection diagrams. For example, in order to define a simple association between two classes, we need to draw three meta-model entities (see Fig. 9, right part). Apart from such pragmatic problems, the approach does not provide for the selection based on indirect relationships and/or name/signature patterns.

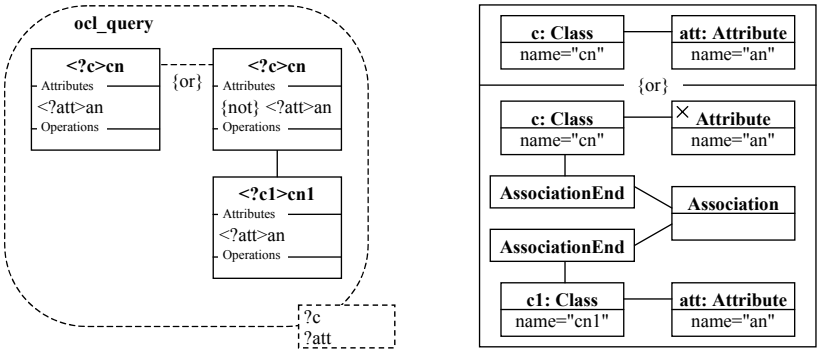

Fig. 9. Representation of the OCL query (from section 2.1). *Left part*: Using a JPDD. *Right part*: Using the notation presented in [12]

5.2 Aspect-Oriented Software Development

In AOSD, a major design issue is where and when to apply crosscutting enhancements implemented by aspects. Aspect-oriented programming techniques provide various textual means to define the conditions under which crosscutting has to take place. Even though various aspect-oriented modeling approaches are around [2] [14] [5], none of them presents a solution to represent such conditions graphically. By means of JPDDs, we now have a graphical notation at hand to visualize the criteria under which a join point is to be enhanced by an aspect. To demonstrate this,

we exemplify in the following how JPDDs may be used to represent pointcuts in AspectJ. We are using the example given in section 2.2[2]:

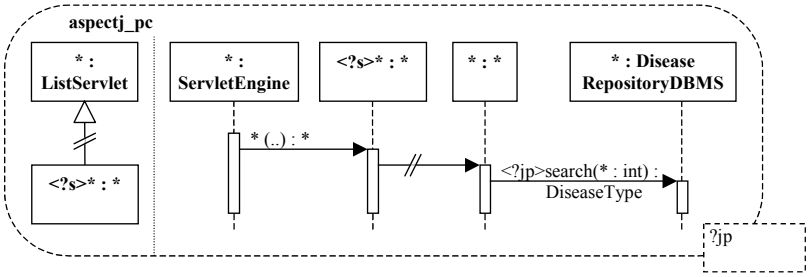

Fig. 10. Representation of the AspectJ query (from section 2.2) using a JPDD

To visualize the pointcut, we draw a JPDD consisting of two parts (see Fig. 10), one specifying the behavioral selection criteria (right part) and one specifying the structural constraints (left part). The parts are linked to each other via identifier "?s". According to the structural constraints, "?s" refers to all children of "ListServlet". In the behavioral part, "?s" is then used to portray a control flow from a "ServletEngine" to one of its children. That control flow must go on, passing any arbitrary number of messages, until an invocation of operation "search" on a "DiseaseRepositoryDBMS" (taking any integer value as parameter and returning a "DiseaseType") is reached. This is the point that the AspectJ pointcut is supposed to retrieve. Therefore, it is given an identifier "?jp", and is placed in the JPDD's template parameter box.

6 Summary and Future Work

In this paper, we presented a graphical notation to define query models. The specification of queries is a new emerging design issue. Queries lie at the heart of novel techniques such as MDA or AOSD. Despite that fact, though, no suitable graphical notation is around to our knowledge that supports the definition of selection queries. An appropriate query modeling language is considered indispensable, though, for techniques like MDA and AOSD to allow them to become widespread. Software developers demand programming language-independent modeling facilities that ease their comprehension on where (MDA- and AO-) refinements actually take place.

In this paper, we have identified frequent selection criteria in query specifications on software artifacts. We presented a comprehensive set of easy to use, yet powerful modeling means to specify such selection criteria in selection queries. We supplemented them with precise OCL semantics that can be executed on existing UML models in order to allocate model elements meeting such criteria. At last, we proved the applicability of our notations with different programming languages using daily-life examples as we can find them in MDA and AOSD.

To the best of our knowledge, our modeling notation is the first approach that provides an essential set of modeling means for the explicit design of queries based

[2] A visualization of the Demeter/C++ traversal strategy presented in section 2.3 is omitted here due to space limitations. Please refer to [15] for a graphical representation.

on the UML. The major advantage of adopting UML's existing modeling means is that query models are easy to write and easy to understand by a broad community of software developers. They do not need to learn a new modeling language, nor do they need to deal with the meta-model (as in other approaches) in order to define query models.

In order to advance the support for software developers in designing queries to a further extent, the following issues are focus of future work: The capabilities allowing the composition of new selection queries from existing ones – like, for example, query aggregations or query specializations – must be improved. Further, suitable abstraction means must be found that enable software developers to reason on selection queries (and their relationships to each other) without bothering about the exact details.

References

[1] Alcatel, Softeam, Thales, TNI-Valiosys, Codagen Technologies Corp, *Revised Submission for MOF 2.0 Query / Views / Transformations RFP*, 18. August 2003

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

[3] Gybels, K., Brichau, J., *Arranging language features for more robust pattern-based crosscuts*, in: Proc. of AOSD'03 (Boston, MA, March 2003), ACM, pp. 60-69

[4] Interactive Objects Software, Project Technology, *Revised Submission for MOF 2.0 Query / Views / Transformations RFP*, 18. August 2003

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

[6] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, Chr., Lopes, C.V., Loingtier, J-M., Irwin, J.: *Aspect-Oriented Programming*, in: Proc. of ECOOP '97 (Jyväskylä, Finland, June 1997), Springer, pp. 220-242

[7] Laddad, R., *Aspectj in Action: Practical Aspect-Oriented Programming*, Manning Publications, Greenwich, 2003

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

[9] OMG, *MDA Guide Version 1.0*, 2003 (OMG Document omg/2003-05-01)

[10] OMG, *Request for Proposal: MOF 2.0 Query / Views / Transformations RFP*, 2002 (OMG Document ad/2002-04-10)

[11] OMG, *Unified Modeling Language Specification*, Version 1.5, 2003 (OMG Document formal/03-03-01)

[12] QVT-Partners, *Revised Submission for MOF 2.0 Query / Views / Transformations RFP*, 18. August 2003 (http://qvtp.org/downloads/1.1/qvtpartners1.1.pdf)

[13] Soares, S., Laureano, E., Borba, P., *Implementing Distribution and Persistence Aspects with AspectJ*, in: Proc. of OOPSLA '02 (Seattle, WA, November 2002), ACM, pp. 174-190

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

[15] Stein, D., Hanenberg, St., Unland, R., *Modeling Pointcuts*, Early Aspect Workshop, AOSD '04 (Lancaster, UK, March 2004)

[16] Stein, D., Hanenberg, S., Unland, R., *A Graphical Notation to Specify Model Queries for MDA Transformations on UML Models*, Workshop on MDAFA 2004 (Linkoeping, Sweden, June 2004)

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